University of Ljubljana
Faculty of Computer and Information Science

Rok Oblak

Web-based simulation and visualization of cardiovascular blood flow

master's thesis

the 2nd cycle master's study programme
computer and information science

Supervisor: doc. dr. Matija Marolt

Ljubljana, 2017

Izvorna koda dela je dostopna na naslovu `https://github.com/oblakr24/simvis-masters`.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Acknowledgments

I thank my family for supporting me for the entire duration of my studies and I thank my mentor Prof. dr. Matija Marolt and dr. Ciril Bohak for their mentorship and guidance.

Rok Oblak, 2017

Contents

List of used acronmys

acronym	meaning
3D	three-dimensional
CFD	computational fluid dynamics
CPU	central processing unit
CT	computed tomography
GUI	graphical user interface
MPI	Message Passing Interface
MRI	magnetic resonance imaging
PET	positron emission tomography
SVVL	Stanford Virtual Vascular Laboratory
TCL	Tool Command Language
VTK	Visualization Toolkit

Povzetek

Naslov: Simulacija in vizualizacija pretoka krvi v ožilju na spletni platformi

Na področju diagnoze krvožilnih bolezni in operacij krvožilnega sistema postaja učinkovita in natančna simulacija krvnega toka pomembno orodje za višanje uspešnosti diagnoz in operacij. Razvili smo celovito rešitev za simulacijo krvnega toka v podanem modelu ožilja in vizualizacijo rezultatov simulacije na različne načine, ki ponujajo informativen prikaz simuliranih količin in s katerimi je mogoča realnočasovna interakcija. Funkcionalnost smo izpostavili preko spletne aplikacije, ki od uporabnika ne zahteva dodatnih namestitev in ki komunicira s strežniškim delom, kjer se izvajajo računsko zahtevnejše operacije. Aplikacija omogoča hitre iteracije delovnega toka in jo lahko zaradi intuitivnega uporabniškega vmesnika uporablja vsak brez dodatne dokumentacije. Poleg tega smo za reševanje problema prenosa simulacijskih podatkov iz zalednega dela do spletne aplikacije razvili metodo kompresije za minimizacijo velikosti podatkov posebej za namen vizualizacije, s katero smo dosegli dobra kompresijska razmerja in omogočili učinkovito shranjevanje rezultatov velikih simulacij.

Ključne besede

tok, simulacija, vizualizacija, krvožilni sistem kompresija, B-zlepki

Abstract

Title: Web-based simulation and visualization of cardiovascular blood flow

In the field of cardiovascular disease diagnostics and cardiovascular surgeries, efficient and accurate blood flow simulation is becoming an important supplementing tool in increasing diagnosis and surgery success rates. We implemented a complete solution for simulating the blood flow in a provided blood vessel model and visualizing the results in a variety of ways, providing informative display of simulated quantities and capable of real-time interaction. We exposed this functionality through a web-based application that does not require any local installations and communicates with a server performing the computationally intensive tasks. It is practical enough to enable fast workflow iterations and can be used by anyone without additional documentation because of its intuitive user interface. We also addressed the problem of data transfer from back-end to client front-end by developing a compression method to minimize the result file size specifically intended for visualization purposes, which was able to achieve good compression ratios and enabled efficient storage of large simulation results.

Keywords

flow, simulation, visualization, cardiovascular system, compression, B-splines

Razširjeni povzetek

Bolezni srca in ožilja so v sodobnem času glavni vzrok smrti v razvitem svetu. Zajemajo več bolezni z različnimi napakami ožilja, ki jih je v veliko primerih mogoče vnaprej predvideti z ustrezno analizo krvnega toka kot rezultata računalniške simulacije v virtualnem modelu ožilja [1]. Take simulacije izrabljajo t.i. računalniško dinamiko tekočin, ki za določen časovni interval simulirajo gibanje toka v modelih, ki so navadno specifični od pacienta do pacienta, ali pa so splošnejši. Te modele je mogoče računalniško obdelati tako, da upoštevajo geometrijske spremembe, ki bi nastale po mogoči operaciji in tako predvideti njeno uspešnost in jo po potrebi prilagoditi. V zadnjih letih je napredek v medicini in računalništvu omogočil vedno naprednejše in natančnejše simulacije z modeli, ki so v določeni meri avtomatsko zgrajeni na podlagi medicinskih slik pacienta. Ko se rezultate simulacije pretoka krvi prikaže na informativen način, lahko to področje precej vpliva na poznavanje srčno-žilnih bolezni in obravnavo vsakega pacienta posebej.

I Sorodna dela

Celoten proces od pacientovih medicinskih slik do analize rezultatov simulacije krvnega toka navadno zajema štiri glavne korake: zajem ustreznih podatkov, specifičnih za pacienta, izdelava grobega modela in mrežnega modela, priprava modelov in drugih podatkov za simulacijo in simulacija sama, ter na koncu procesiranje in prikaz rezultatov [1].

Obstoječi pristopi so se osredotočali na izboljšave enega ali več teh korakov. Zahteven problem je sama izdelava natančnih in realističnih modelov iz medicinskih slik, kjer je cilj čim večja avtomatizacija s čim manj dodatnega človeškega dela. Navadno se pri tem koraku na ročni ali avtomatski način izvaja segmentacija posameznih dvodimenzionalnih (2D) slik in išče zaprte krivulje, ki predstavljajo žilne zidove. Te se povezuje v 3D modele tako, da se krivulje na določeni sliki povezuje s tistimi na sosednjih slikah (zgoraj in spodaj v prostoru). Prva odmevnejša implementacija orodja za avtomatsko izdelavo modelov ožilja je bil Stanford Virtual Vascular Laboratory [2], kjer so se modeli določali s hierarhijo manjših modelov, določenih parametrično. Kasnejše metode so dodale zmožnost avtomatske segmentacije medicinskih slik in tako skoraj popolnoma avtomatiziran proces. Danes je temu namenjenih več močnih orodij, ki nudijo programski vmesnik za različne operacije pri segmentaciji in avtomatski izgradnji modelov in mrež, kot je Vascular Modeling Toolkit [3].

Simulacija temelji na reševanju Navier-Stokesovih enačb [4], ki opisujejo viskozne tekočine kot je kri. Njihova rešitev je hitrostni vektorski tok, s katerim lahko izračunamo ostale količine, ki nas zanimajo; navadno so to predvsem pritiski. Pri reševanju enačb moramo definirati robne pogoje v žilnih zidovih, na katerih je hitrostno polje nično, ter na krovnih površinah vhodov in izhodov. Na vhodih navadno uporabimo analitično metodo [5] za izračun hitrostnega polja na površini vhoda, na izhodih pa lahko določimo upornost, ki modelira odziv preostalega dela žilnega sistema. Danes je mogoče simulirati krvni tok v zelo velikih modelih skoraj celotnega krvožilnega sistema, kot na primeru [6], kjer so avtorji uporabili visoko-paralelno rešitev in lokalno adaptacijo mreže, da je bila simulacija natančnejša na mestih, kjer je to potrebno. Simulatorji toka pa so danes dostikrat izdani kot odprtokodne programske rešitve, kot so Palabos [7], Salifish [8] in OpenFoam [9]. To delo uporablja enega izmed njih, razreševalnik toka v programskem paketu SimVascular [10].

Da so simulacijski rezultati uporabni za analizo, jih je potrebno na ustrezen način prikazati. Najbolj pomembni količini sta hitrostno vektorsko polje in skalarno polje pritiskov. Hitrostno polje se mnogokrat prikazuje zgolj z magnitudami vektorjev ali pa se na podlagi vektorjev zgradi dodatne tipe geometrije. Te nam prikazujejo gibanje toka skozi prostor in vključujejo med drugim tokovnice [11], tokovne trakove ali tokovne cevi [12]. Skalarne enote se prikazuje kot različno obarvane in velike točke v prostoru ali pa na prostorski način [13]. Orodja za vizualizacijo so navadno del specifičnih sistemov za simulacijo tekočin, obstajajo pa tudi prosto-dostopna orodja kot je ParaView [14], s katerim je mogoče vizualizirati podatke v formatu Visualization Toolkit (VTK) [15].

Ker simulacije tekočin pridelajo velike količine podatkov, je pomemben problem tudi njihova ustrezna predstavitev in morebitna kompresija. Pri kompresiji nam je v pomoč dejstvo, da imajo podatki o količinah pogosto neko mero redundance (so medsebojno korelirani) in so le redko podobni naključnim razporeditvam. Med enostavnejše metode kompresije spada kvantizacija podatkov na omejen nabor vrednosti, naprednejše pa poskušajo podatke parametrizirati na različne načine. Med njimi prevladujejo Waveleti [16], s katerimi se navadno kompresira 2D rezine [17] na podobne načine kot počnejo algoritmi za stiskanje slik (JPEG), polinomi [18] ter enodimenzionalni B-zlepki nad lineariziranimi prostorskimi podatki [19]. Vsem je skupen nek način razdelitve prostora, kar je ključno v primeru simulacij krvnega toka v ožilju, ki zajema zgolj majhen del celotnega prostora in je zato potrebna ločena obravnava podatkov v neki okolici.

II Izvedba

Izdelali smo aplikacijo, ki uporabniku omogoča, da naloži model ožilja, določi parametre za ustvarjanje mrežnega modela, določi parametre za simulacijo in na koncu na različne načine pregleduje simulacijske rezultate. Uporab-

niku omogoča tudi nalaganje že obstoječih modelov ter rezultatov prejšnjih simulacij. Sestavljena je iz spletnega dela ter zalednega dela, kjer se nahaja strežnik, ki komunicira s spletnim delom in izvaja nekatere računsko intenzivne naloge v delovnemu procesu.

V zalednemu delu aplikacije smo uporabili ogrodje SimVascular in skripte v jezikih Python in TCL, da smo vhodne modele formata Wavefront .obj pretvorili v format VTK, na njih poiskali in zapolnili vhodne površine ter iz uporabnikovih parametrov zgradili mrežni model, primeren za numerične metode kot je kasnejša simulacija. Ker so vhodni modeli formata .obj, od uporabnika ne zahtevajo posebne predhodne obravnave. Mesta vseh vhodov in izhodov se določijo samodejno, uporabnik pa v spletni aplikaciji določi, katera predstavljajo vhode in katera izhode. Pri izgradnji površin na vhodih in izhodih se izračunajo tudi njihovi približni premeri, s katerimi si uporabnik pomaga pri določanju parametra velikosti roba, potrebnega za izgradnjo mrežnega modela. Ogrodje SimVascular ponuja tudi razreševalnik (angl. *solver*) simulacije tekočin, ki smo ga uporabili za izvajanje simulacije in zbiranje rezultatov.

Na spletnem delu je uporabniku namenjen intuitiven uporabniški vmesnik, s katerim nalaga modele in rezultate, določa parametre za različne operacije ter upravlja s 3D pogledom, kjer so prikazani bodisi vhodni model z določenimi vhodi in izhodi ali mrežni model, bodisi rezultati simulacije. Za določanje vhodov in izhodov je uporabniku namenjen robni pano, kjer so našteti vsi določeni vhodi in izhodi. Ob kliku na vsakega izmed njih se kamera premakne na ustrezno mesto na modelu, kjer je poleg vhoda ali izhoda izrisana še anotacija z njegovim imenom in indeksom. Po uporabnikovem vnosu simulacijskih parametrov preko temu namenjenega okenca se izrišejo simulacijski rezultati, za prikaz katerih smo razvili več različnih načinov vizualizacije:

- prikaz mreže modela s spremenljivo prosojnostjo, ki omogoča pregled drugih količin znotraj modela

- prikaz skalarjev v obliki točk, katerih velikost in barva je skladna z njihovimi vrednostmi; točkam je mogoče z drsniki spreminjati faktor velikosti ter faktor večje ali manjše izrazitosti velikosti glede na njihovo relativno velikost

- vektorski prikaz, primeren za vektorske količine kot so hitrosti in hitrostni odvodi, s katerim so količine prikazane kot prostorsko polje daljic, katerih dolžina in barva odraža njihovo magnitudo

- površinski prikaz, primeren za količine, definirane ob zidovih ožilja, ki glede na velikosti vrednosti določa barvo oglišč mrežnega modela

- prikaz tokovnic za poseben prikaz hitrostnega polja, kjer gradimo povezane krivulje od vhodov modela naprej ter od izhodov modela nazaj s sledenjem smeri hitrosti v prostorski točki z Eulerjevo metodo

Načine je mogoče spreminjati in jim spreminjati parametre na temu namenjenemu stranskemu panoju. V kotu okna je izrisana tudi legenda z barvno skalo in vrednostmi, ki se prilagajajo trenutno izrisani količini. Ker so količine navadno v več časovnih korakih, je trenutno prikazan časovni korak mogoče spreminjati s temu namenjenim drsnikom.

Preden se rezultati pošljejo iz zalednega dela, se nad njimi izvede še kompresija. V ta namen smo razvili in primerjali več različnih pristopov:

- navadna kvantizacija vrednosti z določenim številom bitov na točko, kjer podatke shranimo kot celoštevilske binarne vrednosti in za njihovo dekompresijo potrebujemo le še razpon vrednosti (najmanjšo in največjo vrednost)

- kvantizacija po času, kjer skupaj kompresiramo iste točke v več časovnih korakih in izkoriščamo dejstvo, da se večina točk v zaporednih časovnih korakih spremeni manj kot znaša celotni razpon vrednosti, zaradi česar lahko vsako točko zapišemo z manjšim številom bitov

- kvantizacija s predhodno gradnjo osmiškega drevesa (angl. *octree*), kjer izkoriščamo dejstvo, da se vrednosti v neki okolici medsebojno manj razlikujejo kot znaša celotni razpon vrednosti, zaradi česar jih spet lahko zapišemo z manjšim številom bitov

- hibridna metoda, ki je glavni cilj pri pristopu kompresije podatkov, pri kateri se po razdelitvi podatkov v osmiško drevo te poskuša zapisati kot volumen B-zlepkov in iterativno poskuša z vedno več kontrolnimi točkami ujeti mejo dovoljene povprečne napake, v nasprotnem primeru (če je potrebno več prostora za zapis kontrolnih točk kot za lokalno kvantizacijo) pa podatke kvantizira kot pri prejšnji metodi

Po kompresiji podatkov in njihovemu shranjevanju v binarni obliki se ti pošljejo na spletni del, kjer smo razvili še obraten proces za dekompresijo, v primeru shranjevanja rezultatov pa se ti binarni rezultati tudi shranijo na pomnilniški medij na strežniku.

III Rezultati

V prvem delu evalvacije smo testirali simulacije sedmih različnih modelov z različnimi parametri, vključno z različnimi konfiguracijami vhodov in izhodov. Rezultati so kazali ustrezno odzivnost simulacijskih rezultatov na spreminjanje parametrov, kot so količina in smer pretoka skozi vhode, različne vrednosti upornosti na izhodih in različne postavitve vhodov in izhodov. Vizualizacija rezultatov je dajala informativno sliko, skladno z rezultati zunanjega orodja (ParaView), različni tipi vizualizacij pa so ob primernih vhodnih parametrih kazali zanimive lastnosti kot so vrtinci v toku, vidni na tokovnicah.

V drugem delu evalvacije smo primerjali uspešnost implementiranih metod kompresije. Za testni nabor smo vzeli sedem modelov v štirih različnih nastavitvah:

- 6 časovnih korakov pri vsaj 0.39 % povprečni natančnosti
- 16 časovnih korakov pri vsaj 0.09 % povprečni natančnosti
- 6 časovnih korakov pri vsaj 0.39 % povprečni natančnosti
- 16 časovnih korakov pri vsaj 0.09 % povprečni natančnosti

Rezultati so kazali jasne prednosti kvantizacije po času v primerih z več časovnimi koraki, kot je bilo pričakovano, saj je v tem primeru mogoče bolje izrabljati manjše spremembe posameznih vrednosti skozi čas, vendar pa so bili rezultati te metode slabši pri vrednostih pritiska, ki skozi časovne korake bistveno bolj niha. Uporaba kvantizacije z osmiškim drevesom je bila skoraj vedno ugodnejša od prejšnje in je kazala konsistentno izboljšanje v primerjavi z osnovno kvantizacijo, saj je v povprečju porabila od 30 do 40 odstotkov manj bitov na točko. Glavna razvita metoda, hibridna metoda B-zlepkov in kvantizacije nad točkami v osmiški prostorski delitvi, je bila v vseh primerih najuspešnejša, vendar je bila stopnja izboljšanja precej odvisna od največje dovoljene natančnosti in od kompresirane količine. V primerih kompresije pritiskov je v primerjavi z navadno kvantizacijo v osmiškem drevesu točke zapisala s povprečno 40 do 55 odstotki manj bitov, hitrosti pa zaradi večjega števila izstopajočih točk (ki negativno vplivajo na ustrezno parametrizacijo z volumni B-zlepkov) z med 6 do 25 % manj bitov. Metoda je uspešnejša pri manjši ciljani natančnosti, saj je v tem primeru regresija z volumni B-zlepkov lažja.

IV Struktura

Predstavitev problema ter pregled sorodnih del in obstoječih rešitev je v poglavju 2. V poglavju 3 predstavimo implementirane metode kompresije rezultatov simulacije toka.

Poglavje 4 v več sekcijah opisuje implementacijo, vključno z uporabniškim tokom (podpoglavje 4.2), zalednim delom (podpoglavje 4.3) ter spletno aplikacijo in tipi vizualizacij (podpoglavje 4.4).

Pregled rezultatov in evalvacije ogrodja je predstavljen v poglavju 5, v poglavju 6 pa so naši zaključki, ugotovitve in predlogi za nadaljnje delo in izboljšave.

Chapter 1

Introduction

Cardiovascular diseases are the primary cause of deaths in the modern developed world. They encompass many conditions where there is any kind of cardiovascular system malfunction caused by the heart or the vascular system itself. Many of these conditions can be predicted, explained or identified in an early stage by examining the blood flow dynamics causing various loads and stresses to the cardiovascular system [1].

The cardiovascular system is a complex physical system that has seen many attempts to describe using mathematical models, but in recent years the most efficient ones deal with directly simulating the blood flow in the veins and arteries. These belong to the field of computational fluid dynamics (CFD) simulators and have been used successfully for many years to try to simulate the accurate conditions inside the cardiovascular system so that they can be studied by medical personnel. They include the generation of patient-specific models of cardiovascular structures, construction of hypothetical vascular structures to test their flow dynamics, and the combination of both by examining the implications of various possible surgeries changing the vascular structure of a specific patient.

Afterwards, the results can be analyzed and visualized to provide physicians valuable insights into cardiovascular diseases they could not otherwise

obtain. Virtually simulating blood flow is also both non-invasive and does not require the presence of a patient, can be performed many times with different parameters and can provide much more detailed data about flow dynamics. Historically, virtual blood vessel models have been created from manually observing organs of deceased subjects, but modern advances in imaging technologies have enabled scans of live patients to be used for patient-specific model generation, significantly expanding the applications of blood flow simulations. For example, a stent (a mesh tube inserted in an artery) procedure or a bypass surgery (the flow is diverted from an artery through a different vessel) are procedures that involve significant changes to blood vessel geometry and incur significant risks because of various unknowns in hemodynamics if no prior analysis is performed. A patient-specific approach would approach this issue by creating a virtual model of the proposed surgical changes to an existing model and performing a blood flow simulation to accurately predict the changes in blood flow visible after analyzing and visualizing the results. This could then be used to either confirm the proposed procedure or seek another one, minimizing the risks and increasing the surgery's effectiveness rate. A software framework can therefore offer a lot of potential in providing various functions for virtual patient-specific blood flow simulations.

The topic of this thesis is a software application focused on ease of use and fast workflow iteration, where the user can quickly and effortlessly come from a basic model to visualized simulation results which are informative and interactive in real-time.

1.1 Structure

In chapter 2 we give an overview of the problem domain, the related work on flow simulation frameworks for medical uses, as well as an overview of flow simulation compression problem and its related approaches. Chapter 3 discusses our approach to compressing the flow simulation result data by

implementing different compression methods. Chapter 4 describes the implementation and is split in several sections. Section 4.2 presents the general workflow of application usage from the user standpoint. Section 4.3 discusses the back end implementation while section 4.4 presents the front end implementation, architecture (subsection 4.4.1) and various types of visualizations implemented (subsection 4.4.2).

The application usage and comparison of implemented compression methods are evaluated in chapter 5 while chapter 6 presents our conclusions and findings, the work's contributions and proposals for further improvements.

Chapter 2

Related work

2.1 Flow simulation and visualization

The complete process from taking patient observations to analyzing the simulated blood flow results typically involves four major stages: data acquisition where patient-specific data is gathered, model generation where meshes are created from the gathered data and prepared for simulation, the simulation itself and lastly, results gathering and postprocessing [1]. Every step is important by itself and determines the performance of the following steps, which is why different approaches have focused on different steps of the whole process.

2.1.1 Model and mesh generation

Data is usually collected using high-precision magnetic resonance imaging (MRI [20]), computed tomography (CT [21]) or positron emission tomography (PET [22]) scans and then processed using a mix of manual and automatic methods involving specialised techniques to segment the stack of medical images into regions and connect them along the stack to produce three-dimensional (3D) geometry. In case of 3D vascular mesh generation, either manual paths and lofts are created, or the mesh is automatically seg-

mented from its background using image processing techniques. The first method usually involves contour drawing on 2D segmentations which are then automatically lofted into tubular sections and joined together via boolean operations to produce a rough joined model [23]. The model usually requires a significant amount of manual tweaking to ensure smooth edge transitions and local accuracy, which is why this method can last significantly longer. However, a certain degree of manual work is usually preferred since it is important to have accurate and smooth surfaces to achieve realistic simulation results, so the extra time needed is usually not a deciding factor, especially if the model is expected to be used in multiple simulations and across multiple applications.

Stanford Virtual Vascular Laboratory (SVVL) was one of the first attempts to model blood flow in blood vessels by developing a software framework integrating model construction, mesh generation, flow simulation and visualization [2] . The framework introduced a knowledge-based engineering approach to build a more complex vascular model from a hierarchy of simpler primitives defined parametrically. The model is then meshed with help from the user. It regarded the blood flow as an incompressible Newtonian fluid in rigid vessel walls, and reduced it to incompressible Navier-Stokes equations. These equations are given suitable boundary conditions specifying velocities on certain surfaces (vessel caps) and tractions along vessel walls, among others, in order to be solved numerically by a flow-solver. The results were collected at pre-defined nodal points and visualized in different ways, including flow vector fields and streamlines.

A major shortcoming of SVVL was its inability to generate patient-specific models from medical images, which was addressed by [24], one of the first attempts to reduce the time needed for patient-specific model generation by automating the majority of the process. It performs automatic level-set segmentation of 2D slices, creates a model using lofting operations and meshes the model into a simulation-suitable mesh.

In recent years, various other, more sophisticated approaches have been developed for user-guided modeling and meshing from medical image data, such as the Vascular Modeling Toolkit [3], a framework which provides powerful functionality like image segmentation from medical images to generate meshes, vessel centerline (a curve passing through the center of a branch) extraction to determine the model topology, subsequent mesh generation and various post-processing techniques to ensure optimal mesh quality at the correct locations.

2.1.2 Flow solvers

The simulation itself is a task of solving Navier-Stokes [4] equations describing the flow of viscous substances, in our case human blood. The solution to these equations is a flow velocity field, which serves as a basis for calculating other physical quantities such as pressures or temperatures, with former being of most interest in blood flow simulations.

In the domain of blood vessel flow simulations, Navier-Stokes equations are solved with given boundary conditions specific to blood vessels and blood properties. The boundary conditions are defined by vessel walls, where the flow cannot pass through and the resulting condition is a zero-velocity surface with additional properties such as traction alongside the surface, as well as by the surfaces closing the vessel walls, or "caps", defining flow inlets and outlets. An inlet surface needs to define a boundary condition varying in time to simulate a smooth, periodic flow rate, and is usually as a velocity surface which can be calculated analytically from input time-dependent flow rates, as shown by [5]. The simulation length and the density of captured results can be set to capture certain events during one or more cardiac cycles where the flow rate is usually "pulsatile" or responding to cardiac cycles, but it can also be constant or "steady" if we are more interested in stable flow vector fields in a given vascular mesh.

The outlets usually define a pressure or a velocity surface boundary con-

dition or more complex ones such as resistance or impedance boundary conditions, simulating the response of downstream vasculature (the part of the whole vasculature not being simulated, located downstream from the outlet surface).

Software CFD solvers have been used to simulate blood flow for many years, initially in two dimensions and later in three as technological progress enabled more detailed and faster simulations. Today, flow simulations of the entire vasculature can be performed, provided enough time and computing power, as demonstrated by [6] where the authors used a highly parallel solution on very large meshes, using anisotropic mesh adaptation to achieve optimal local density, on thousands of CPU cores.

Many commercially available or open source CFD solvers exist today, such as Palabos [7], using the Lattice-Boltzmann method, a molecular analogue of the Navier-Stokes equation, to achieve high accuracy and efficiency. Several other open-source frameworks are also available, such as Sailfish [8] and OpenFoam [9], as well as solvers included in frameworks specifically targetd for blood flow simulation, such as SimVascular [10] (described in subsection 4.1.1), used in this work for simulation and meshing purposes.

2.1.3 Visualizing the results

Visual representation of flow simulation results is an important aspect of post-simulation analysis and gives an overview of the data as a whole, as compared to certain metrics that only measure specific aspects and thus cannot give a complete picture individually. From a user perspective, the two most important quantities are flow velocity, represented as a 3D vector field, and flow pressure, a 1D scalar field. They describe flow movement through space (and time), where vortices form, how the flow responds to branching (and confluences), and stresses are induced on the vasculature due to local pressure differentials, etc. Other derived quantities are also informative to for medical uses, such as shear stresses along vessel walls (wall shear stresses)

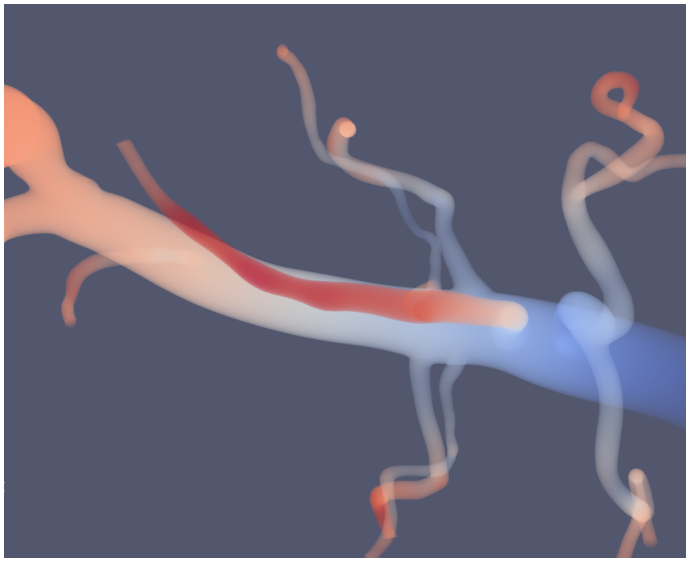


Figure 2.1: Volumetric rendering of flow simulation data in ParaView. For this data this method is unfortunately too slow to be used in real-time.

indicating the stresses acting parallel to mesh surfaces, in-plane tractions, etc. These are usually displayed on the mesh itself, since they only concern its surface and not the inside volume.

Scalar values, such as pressures or velocity magnitudes, are usually displayed by sprite points or 3D spheres colored and scaled proportionally to their values, or with volumetric rendering. The latter is especially common for displaying medical images from MRI [13] or CT scans since it is the most basic way of visualizing multiple 2D slices without the need of additional segmentation. This method is usually performed by casting rays from the camera to each image pixel and sampling the data in the ray-volume intersection and is usually very computationally intensive which presents a problem when rendering large simulation results in real-time.

Velocity vector fields can also be visualized with streamlines and other similar or derived methods [11] which display flow movement through space (streamlines) or time (streampaths). Streamline construction can be signif-

icantly faster when using correct cell-based representation, as shown by [12] which developed a special RK-4 method for streamline, streamribbon and streamtube construction.

Visualizing the simulation results with commercially available software usually presents some limitations due to unconventional file formats used to store the result data and often requires additional result manipulation and formatting (as in our case), but there are separate software applications intended for general scientific visualization, such as Paraview [14], a desktop application for Visualization Toolkit (VTK) file formats [15], alongside its web-based implementation (which still requires a separate server installation), ParaViewWeb [25]. In our case, ParaView was used to validate the simulation and visualization results as our result data file formats were also in VTK formats, since ParaView it also includes some of the same visualization options as developed for our solution. An example of one of our simulation results rendered in ParaView is seen in figure 2.1.

2.2 Compression

2.2.1 Motivation

Flow simulations usually produce results in the form of scalar values for different quantities across the sample points in space. The number of these points can be in hundreds of thousands and the number of quantities measured can be large, as can be the number of time steps where the scalar values are recorded. This can result in large file sizes if the results are stored arbitrarily, e.g. without any compression and perhaps even in a file format with an additional overhead.

For example, a model with 500 000 sample points, 30 time step records for the quantities of pressure and velocity (composed of three components for every direction) in double precision yields a file size of almost half a gigabyte.

This is not very suitable for storing many result sets on the disk, and also not suitable for transferring these files over the network. Both constraints apply for our case - the results set can be saved after every simulation for later use, and every time the user requests them the files need to be transferred via the network from our server to the user's browser. Without compression the server application would quickly fill with gigabytes of result files and the user would have to wait additional time after every simulation or the request to load previously-saved results.

Fortunately, we can make use of our specific problem domain to tackle the issue of simulation data compression. First, as our result data is intended for visualization purposes and not for any additional scientific analysis, the required precision is substantially lower than the IEEE 754 double-precision binary floating-point format [26]. In fact, as will be discussed later, it is sufficient to encode the data with 256 or fewer different values - as long as these values correctly map to the original data range.

Furthermore, we can exploit some other features of our specific problem domain. In smaller scales (in time and space) the resulting values are not "chaotic" in nature, meaning that there is locality where local points are "similar" - they occupy a smaller data range than the global data range. For example, if a global data range ranges from 0 to 100, a certain point neighborhood only ranges from 20 to 30, and another from 50.00 to 50.01. This locality exists in both the time and space domain and both can be used for our purpose.

2.2.2 Related work

Medical images and general medical digital data, including flow simulation data, have seen many different compression approaches. They usually attempt various flow-field parametrization and spatial subdivision approaches. Spatial subdivision is not required for medical images since they usually take an entire cuboid volume, but is important for compressing flow simulation

data where the flow is limited to a vascular mesh and it is sensible to treat it individually in different places.

The approaches differ in their targeted accuracy - for example, [19] aims for high-accuracy compression by linearizing the data points, sorting them and fitting a precise one-dimensional B-Spline. This approach requires additional bits per data points to store indices to recover the data after sorting, which is acceptable in this case as the targeted error rate is very low and the total number of bits per data point can therefore be somewhat larger.

Lossless methods have also been proposed, such as lossless stationary wavelet transform on 2D slices of 4D medical images (volumetric data through time) [17], where an integer wavelet transform is used to remove spatial redundancies and a lifting wavelet transform is used to remove temporal redundancies. Wavelets [16] in general are often used for medical image compression and often follow similar processes as employed in classic JPEG image compression or MPEG in case of temporal data in order to decrease temporal redundancies. Wavelets have also been used for flow simulation data, such as [27] where a discrete wavelet transform was used on an octree subdivision of airflow simulation data.

Other types of regressions have also been described, such as using polynomials [18] on rectangular blocks of 2D medical images and encoding the polynomial coefficients with Huffman coding, and also encoding the residual error with run-length encoding to achieve lossless compression.

2.3 Our approach

The purpose of this thesis was to create a framework for blood flow simulation and visualization. The framework would be composed of a web-based front-end and a back end for performing the most computationally intensive workloads. It would be easy to use, enabling a user without any previous experience with CFD software to perform both the simulation and visualize the

results, as opposed to existing solutions requiring special software and complicated user interfaces. The front-end would be web-based, freeing the user from having to install any additional hardware, whereas the computationally-intensive simulation would run on a server, giving the user more flexibility and simulating the results in a shorter time, while also providing the ability to use and save existing models and result sets. The blood vessel models required for the input would require as little pre-processing as possible and be in a simple format as possible. The visualizations would be informative and comprehensive, giving the user a complete picture of the simulation, as well as interactive in real-time.

Chapter 3

Compression of flow simulation data

We implemented and evaluated several different approaches to minimize the file size of simulation result data. This chapter discusses the individual methods and their features.

3.1 Methods

In the following sections we present several different compression approaches we implemented. We tried to do fair comparisons and point out the pros and cons of each approach. The starting point is a fixed maximum average error rate that we set for every approach. Experimental tests determined different average error rates most suitable for different approaches.

The error of a single record is defined as the ratio of the difference between an encoded value $encV_i$ and the real value V_i and the data range: $\frac{|V_i - encV_i|}{range(V)}$. In addition, since we have many sample data points we can expect some to be "outliers", e.g. their values are either much higher or much larger than most other values - this can be an expected situation in flow simulations where many chaotic regions can occur and a lone randomly sampled point can take

a seemingly unnatural value. This presents a problem when visualizing these data points since such outliers can, for example, skew the color palette and make most values hardly discernible among each other. For this reason, it is reasonable to instead take a shorter data range, taking the range of p % values and discarding the $(100 - p)$ % outliers. For example, for $p = 99.9$ %, we would sort the values and discard the top and bottom 0.05 % values. This usually produces a substantially shorter data range, sometimes more than half as short. It is worth mentioning that making the global data range shorter by discarding the outliers is detrimental to the performance of our compression methods since they have a stricter accuracy constraint to satisfy.

When encoding the values with b bits, we are left with $2^b - 1$ bins where each value is rounded to its nearest bin; for mostly evenly distributed values the largest error is then one half the bin width, while the average value is one quarter of the bin width. The defined error threshold with b-bit encoding is then defined as:

$$e_{thr} = \frac{0.25}{(2^b - 1)}, \tag{3.1}$$

which will serve as an upper average error threshold for other methods. For example, this produces an average error rate of 0.09 % for 1 byte and 0.397 % for 6 bits.

3.2 Basic quantization

The first and most basic way of compressing a certain quantity is to use a fixed amount of bits to encode every data point record (a single scalar value) by means of quantization - linearly mapping the values from a very large set (single to double precision floating point accuracy) to a much smaller set of values defined by a limited number of bits. This way, to be able to recover (decompress) the quantized values, the only other information we need is the data range of the quantity - its minimum and maximum values.

Given the minimum and maximum values V_{max} and V_{min}, the value val is encoded with b bits as

$$V_{enc} = |(2^b - 1)\frac{val - V_{min}}{V_{max} - V_{min}}| \tag{3.2}$$

and decoded as

$$val = \frac{V_{enc}}{(2^b - 1)}(V_{max} - V_{min}) + V_{min}, \tag{3.3}$$

where a certain error is introduced by the rounding, its average calculated by (3.1).

Assuming the point positions are already broadcast, the entire record for a single quantity would then be the minimum and maximum values, encoded in a standard IEEE 754 single-precision floating point format [26] (8 bytes in total), followed by the bit number information (itself fitting in a number of bits, or it could be skipped if this information is known implicitly) and then followed with a series of b-bit blocks for every data point.

It is obvious that for records with many data points the constant factor becomes negligible and the number of bits per data point is close to b. As stated earlier, the total data range taken is taken by discarding a set percentage of outliers and is therefore somewhat smaller. This compression method will serve as a baseline for evaluating other methods' performance.

3.3 Time-domain quantization

The second approach tries to exploit the time-domain locality of values. Since we often want the simulation data in several different time steps (as opposed to, say, just at the end) and these time steps are not spaced too far from one another, we can expect that a single data point will not fluctuate too much in time, at least as opposed to every other data point. We can therefore expect that the data range of a single data point in time is shorter than the global data range. The global data range, in this case, includes all the points

across all time steps and can therefore be larger than the data range of all the values in a single time step.

This way, we can quantize the values in the time domain just like in the previous approach. After taking the initial value at a "keyframe" time step t_{K1}, $val(t_{K1})$, the following values up to the value at the next keyframe time step t_{K2}, $val(t_{K2})$, can be encoded (and decoded) with b bits in the same way as in basic quantization (3.2), where $val(t_{K1})$ and $val(t_{K2})$ are taken as V_{min} and V_{max}, respectively.

The process is repeated for t_{K2} and t_{K3}, t_{K3} and t_{K4} and so on. The values at keyframe steps are themselves quantized to save additional space. The determination of keyframe steps can be done by selecting time steps where the slope changes the most, but it usually suffices to just take a single first and last keyframe, only performing the process once for a single data value.

The issue in our case is that for the quantities of pressures and velocities the values at the first time step are usually very low or zero, since they only depend on initial conditions set by the user (the initial pressure and flow velocity at the caps). The values then change rapidly in the next time step and then proceed to change less in subsequent time steps. For this reason we encode the first time step values in a classic manner as described in the previous method, and then separately encode the rest of the time steps with keyframes determined among time steps t_2 to t_k where k is the total number of steps. Additionally, every data point now needs to specify the number of bits taken by the values in time, since this number can differ between data points. This requires 4 bits to specify the bit number (since we have an upper bound on the number of bits per point).

The advantage of this approach is that since most values take smaller data range in the time domain, they can be encoded with fewer bits than in the previous approach, while still maintaining the required accuracy constraint. The average ratio of local time-domain data ranges to the global data range

depends on the quantity and is about 20-50 % in case of pressures (meaning 1-2 fewer bits are needed to encode the values) and 2-15 % in case of velocities (meaning 3-6 fewer bits are needed).

The main disadvantage of this approach is its dependence on the assumption that values don't change much over time and its sensitivity to having many time step records. There can be cases where values change more in time, for example in case of pulsatile flows and in case where the neighboring time steps are further apart if the results were not collected at every time step. After all, a user might be interested in more change over time and might want a longer simulation with more spacing between result time steps, as opposed to shorter spacing where time steps are close together. Also, in case where there are only a few result time steps gathered, the overhead incurred by the bit number and the keyframe values can stop being negligible and can make the overall performance worse than the basic quantization method. This can often occur when encoding pressures since they can widely vary through time, but is less of a problem with encoding velocities, as will be seen in the results section (5).

3.4 Octree quantization

This approach tries to exploit the space-locality of data points. Since we expect neighboring data points to take a smaller data range than the global data range, we can subdivide the space into smaller regions and treat each region independently. This method uses the octree [28] subdivision method to subdivide the space into rectangular cuboid shaped regions, as seen in figure 3.1. The tree is built by depositing into it data point positions, where the initial space - the root node - is subdivided into eight equally sized subregions - its child nodes - and then recursively subdivided further until the specified depth is reached, where the data point is deposited to the node's data, a list of "payloads" . Every payload contains the point coordinates as

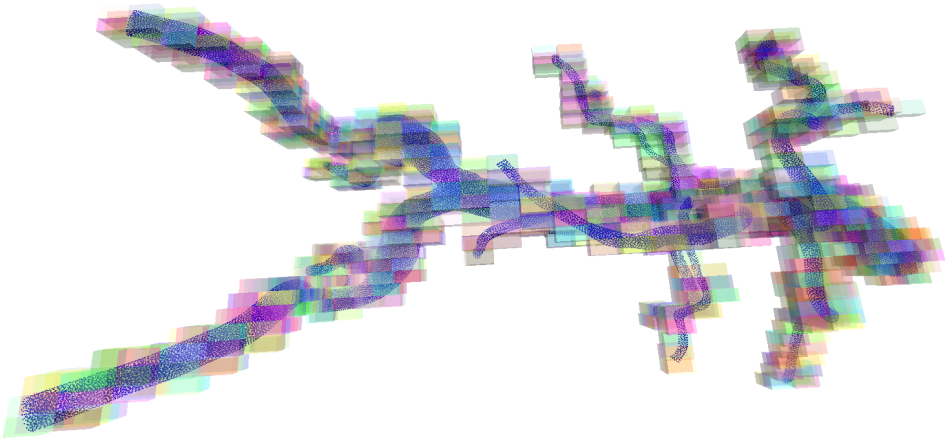


Figure 3.1: An octree decomposition of data points on a model of aorta.

well as its index so that we can map the point to its respective quantity.

The encoding process then encodes each node with the minimum and the maximum values of the data referenced by its payload indices, as well as a series of data values, each encoded with b number of bits. Corresponding to the local data range, the same formula (3.1) applies for selecting b. This parameter must also be recorded in the resulting file, taking 4 bits.

The tree is built with a constant depth, but since we keep the list of payloads in every node, including non-leaf nodes, it is trivial to collect the regions at a certain depth. Our experiments showed that the depth choice is important - too deep, and most nodes will only hold very few payloads, too shallow, and most nodes will take too large data ranges. Nodes with few payloads incur too big an overhead because of the minimum and maximum values as well as the bits number value. Nodes holding too many payloads tend to take large data ranges and therefore cannot encode the data much more efficiently than the basic quantization method. In the end, we chose the depth where the average number of payloads in the nodes was around 1000. This way, most nodes were "full", (their bounding volume was entirely

inside the mesh), and still took small data ranges, while the ratio of nodes with few payloads was small enough for their overhead not to matter too much in the resulting file size.

3.5 B-Spline regression

The inspiration for this approach was the fact that the resulting quantity values often seem like they follow a certain analytical function, e.g. the values transition in smooth curves. This is often visible if only a limited local region is considered. An example of smooth value transitions is seen in figure 3.2, in this case in a 2D horizontal slice. To get an initial evaluation on possible parametric curves the data values could be regressed to, we first looked at one and 2D data by taking a line and plane intersects of the data points. Since the data points are discrete, we had to perform the same kind of interpolation as we did in case of streamline generation (equation 4.3) to get an approximate value at any position in space. These intersections show obvious curves formed by the data. If we could parametrize an approximation of these curves, the parametrization might be expressed in a smaller number of bits than other approaches where each data point is recorded separately (with a certain number of bits).

Since the data usually only follows these curves in local regions and not on global scale, it is necessary to perform spatial decomposition and treat each partition separately. This is also required because data point positions are not spread evenly in space which, since they are confined in blood vessel walls, is mostly sparse. It is therefore important to separately treat points in different vessel branches since they are (to a point) independent physical systems. Therefore, attempting to find a single parametric solution for several of them would be inefficient. Blood vessel geometries can have many small branches and can be particularly sensitive to spatial decomposition.

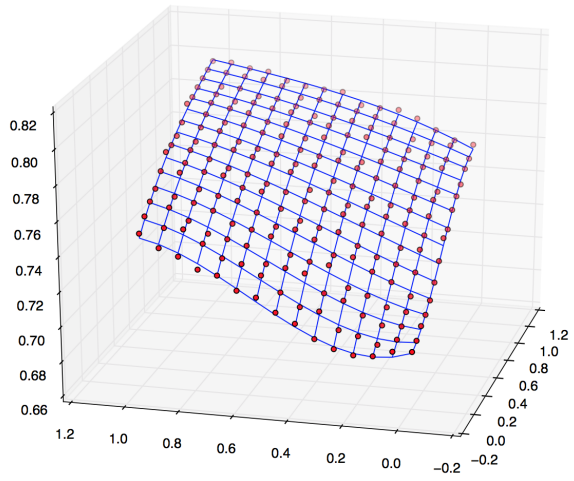


Figure 3.2: An example of a B-Spline surface with 16 control points regressed to fit 225 pressure data points of a 2D horizontal slice, with average and maximum errors of 0.15 % and 0.54 %, respectively.

3.5.1 B-Splines

We chose B-Splines to parametrize the values approximation because of their good ability to fit along their control points. A B-Spline curve of degree k is a generalization of a Bezier [29] curve defined on a knot vector

$$T = (t_0, t_1, ..., t_{k-1}, t_k, t_{k+1}, ..., t_{n-1}, t_n, t_{n+1}, t_{n+k}).$$

In our case, we choose a non-periodic (the first $k+1$ knots are equal to 0, the last $k+1$ knots are equal to 1) and uniform (internal knots t_k to t_n are equally spaced) B-Splines for practical and convenience purposes. Given the B-Spline basis functions defined with a recursive formula:

$$N_{i,0}(t) = \begin{cases} 1, \ t_i \leq t \geq t_{i+1} \\ 0, \ otherwise \end{cases}$$

$$N_{i,0}(t) = \frac{t - t_i}{t_{i+k} - t_i} N_{i,k-1}(t) + \frac{t_{i+k+1} - t}{t_{i+k+1} - t_{i+1}} N_{i+1,k-1}(t),$$

a one-dimensional B-Spline curve is defined as

$$S(t) = \sum_{i=0}^{n} N_{i,k}(t)\, p_i,\ n \geq k-1,\ t \in [t_{k-1}, t_{n+1}],$$

where p_i is the i-th B-Spline *control point*, which controls the local curve behavior. In our case, the data is three-dimensional, so the B-Spline is parametrized along three directions, producing a B-Spline volume [30]

$$S(u,v,w) = \sum_{g=0}^{n}\sum_{h=0}^{n}\sum_{i=0}^{n} N_{g,k}(u)\, N_{h,k}(v)\, N_{i,k}(w)\, p_{g,h,i},$$

where u, v and w are our point coordinates and $p_{g,h,i}$ are the control points in a cubic formation.

From this equation we can rearrange the basis function outer products in a single three-dimensional matrix:

$$\boldsymbol{x} = [N_{0,k}, ..., N_{n,k}(u)] \otimes [N_{0,k}(v), ..., N_{n,k}(v)] \otimes [N_{0,k}(w), ..., N_{n,k}(w)],$$

where $\otimes$ is the outer product of two vectors. Having M values we can express this as a linear system $X * P = B$ where X is a matrix of size $M \times n^3$ filled with rows of matrices x flattened into vectors of size $1 \times n^3$, P is a vector of control points of size $1 \times n^3$ and B is a vector of quantity values of size $1 \times M$. Since P is an unknown, we can find a best fit solution $P = (X^T X)^{-1} X^T B$ using linear least squares [31] to minimize the sum of squared differences. The equation is solved using a least squares solver provided by numpy.

Calculating the basis function coefficient along the spline is computationally intensive, so the results can be cached by pre-calculating the values in a finite number of inputs and storing them in a matrix of size $n \times N_S$ where N_S is the number of approximation samples in a linear range $[0, 1]$. A suitable approximation of any input is then taken as a linear interpolation between the two closest pre-calculated values.

3.6 Hybrid approach

Our approach combines the approaches of space-decomposition quantization and B-Spline regression. It performs the same octree spatial decomposition as in section 3.4 and individually compresses each partition. In doing so, it selects the most suitable of the two methods by constraining the average error rate and comparing the number of bits required to encode the points in the partition.

The method iteratively attempts to perform B-Spline regression to encode the data by going through a list of "presets" defining the number of B-Spline control points N_c in one dimension, starting with 2 (8 total) and going through to 9 (729 total). After every iteration, the average error level is measured and the process is repeated with the next control point preset until the measured error level meets the locally calculated criterion (3.1); this way a suitable B-Spline solution is found. The control points are converted to IEEE 754 half-precision floating point format [26] (16 bits) prior to testing the error level in order to save space. This can sometimes lead to improper solutions because of accuracy loss but such cases are rare enough in our testing to make 16-bit format preferable to the standard 32-bit single-precision format.

Otherwise, the iteration stops when the number of bits required to encode the control points ($16N_c^3$) is larger than the number of bits required to do local quantization (bN where b is the number of bits per point and N is the number of values in an octree block), or if the iteration process does not find a suitable preset meeting the error criterion. If the local value range $|V_{max} - V_{min}|$ is smaller than the global desired error criterion, the values are encoded as a constant, implicitly set to be the midpoint between the minimum and the maximum values ($\frac{V_{min}+V_{max}}{2}$).

Each region is encoded in a binary format with the following information: compression type used for this region, minimum local value V_{min}, maximum

bit block num.	1	2	3	4	5	6	...
B-Spline	type (2)	V_{min} (32)	V_{max} (32)	N_c (5)	c_1 (16)	c_2 (16)	...
quantization	type (2)	V_{min} (32)	V_{max} (32)	b (4)	v_1 (b)	v_2 (b)	...
const. enc.	type (2)	V_{min} (32)	V_{max} (32)	-	-	-	-

Table 3.1: The binary file format of a single octree block, depending on the type of encoding used for this block (B-Spline regression, quantization or constant encoding).

local value V_{max}, followed by data depending on the type of compression used, as shown on table 3.1. The constant encoding does not need any further data, while the other two types need the number of control points along one dimension N_c and the control points c_i themselves for B-Spline compression, or the number of bits per point b and quantized values v_i for quantization. Also required is the binary information about the selected octree depth (3 bits), format of the data (1D scalars or 3D vectors, 1 bit) and the number of result steps (8 bits), stored before the blocks at the beginning of the file. This way the decompression algorithm can correctly recover the individual blocks and decompress the data.

Since the method works by iteratively comparing the suitability of both methods, it almost always finds a better solution than any of the single methods by themselves. Its only added overhead is an extra type bit per each partition. In practice, the choice of data and the selected quantity, the model geometry and the desired maximum average error rate determines the ratio of partitions encoded with each compression type. However, we also noted that when a B-Spline solution is used in a certain octree block, it typically uses considerably fewer bits than quantization. More on the comparison will be discussed in the results chapter (5).

The downside of this method is its sensibility to outliers, as will be discussed in chapter 5, which determines the amount of regions suitable for

B-Spline regression, and its performance. Currently, since several iterations are usually required for each region, and because several matrix operations are performed on the CPU, this method is slower than other methods. In future work, this could be mitigated by tweaking the iteration parameters and migrating the code to a graphics processing unit (GPU) to speed up matrix calculations.

A separate decompression module was developed on the front-end which performs a reverse operation to decompress the binary files. Since point positions are transferred separately, the octree structure can be recreated without requiring any additional data.

Chapter 4

Implementation

4.1 Technologies used

The following is a list of technologies used for the server-side application and the web application.

4.1.1 Back end

TCL

Tool Command Language (TCL) is a dynamic programming language primarily intended for server-side scripting. It is used for generating the custom meshing and simulation scripts used by SimVascular commands and executables. These TCL scripts are executed in the SimVascular TCL shell.

VTK

Visualization Toolkit (VTK) [15] is an open-source software package for computer graphics, visualization, volumetric methods and mesh processing and many other techniques. It is used in our back-end pipeline for storing models (after being converted from Wavefront .obj [32]) and results (before being compressed) in the VTK polydata and unstructured grid formats. VTK

Python wrappings are used in Python scripts to process these VTK files, perform various mesh operations on the models, extract result data, etc.

Node.js

Node.js [33] is an open-source server-side JavaScript runtime. It enables fast and efficient server-side scripting and has an extensive package management system, NPM, providing many useful back-end utilities. It is used for the main server-side application logic to perform the file system operations, execute the scripts, interface with the front-end and provide the web application to the user.

Numpy

Numpy [34] is a scientific and numerical computing package for Python primarily intended for matrix operations in linear algebra. It is used to efficiently store the result data and perform algebraic operations on matrices and vectors in the compression module.

SimVascular

SimVascular [10] is an open-source software package providing several tools in the pipeline of medical image data segmentation, mesh generation and simulation. It is composed of several tools packaged in a user interface enabling the user to manually import medical images, create the blood-vessel model and process it, as well as perform the simulation. Fortunately for us, it also provides a limited functionality through command-line commands, exposed through its TCL shell or separate executable commands. The following two components packaged in SimVascular were also used for our purpose:

TetGen

TetGen [35] is a tool for generating Delaunay tetrahedralizations on a 3D mesh to convert an input 3D model and create a volumetric tetrahedral mesh suitable for numerical methods such as the finite-element solver used for the simulation. Developed by the Weierstrass Institute for Applied Analysis and Stochastics, Germany, it is free to use and exposed through the SimVascular TCL shell.

SVSolver

This is a finite-element CFD solver evolved from PHASTA [36] and exposed as a standalone executable in the SimVascular package, and is a part of two other executables, the pre-solver and the post-solver. The solver is used to perform the simulation after files characterizing the finite element solution are created by the pre-solver which can be thoroughly configured using script files specifying parameters, boundary conditions and initial conditions. The data generated by the solver is parsed by the post-solver to generate the result files, characterizing the finite element solution in several time steps in a defined time period.

MPICH [37], a message passing interface implementation, is used with the svSolver command for distributing the simulation workload across multiple processor cores or on a computer cluster, which can substantially increase simulation speed.

4.1.2 Front end

Angular.js

Angular.js [38] is an open-source JavaScript front-end application development framework developed by Google. It was used to develop our front-end application and provides a suitable framework for creating model-view-

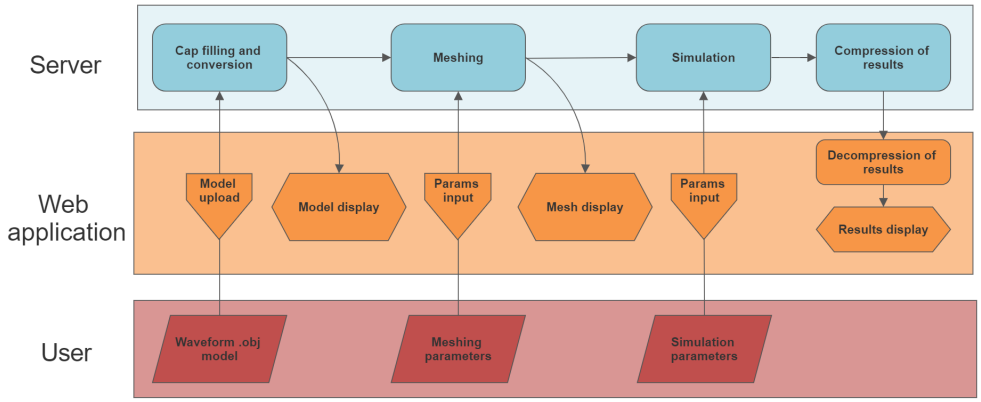


Figure 4.1: A diagram of data movement in the entire application workflow with user interacting with the web application and the web application interacting with the back-end.

controller application architecture and various useful functions such as dependency injection and dynamic HTML creation.

Three.js

Three.js [39] is a cross-browser JavaScript API using WebGL to render 3D graphics content in a browser. It enables a developer to write very efficient and performant code executed on the GPU to render complex graphics content and animations with minimal browser overhead. It is used on our web application to render the main 3D canvas and its content - the mesh and various result visualizations.

4.2 User workflow

The application targets a user with knowledge of the problem domain (presumably with a medical background) but not necessarily having any experience with computer blood flow simulation. For this reason the user interface

is kept simple while still preserving all the necessary data entry points to produce a full-featured flow simulation. The workflow is presented in figure 4.1 and was roughly modeled after that of the SimVascular desktop graphical user interface (GUI) application.

4.2.1 Model upload

For our app, we chose to simplify the model submission by requiring that the format be the Wavefront .obj, the most common and simple format for 3D models and one that any other format can easily be converted to. We require that the model contains holes where cap faces are located. This way it is easier to produce the models since the cap faces do not need to be created and manually annotated. Instead, the cap faces are created automatically on our back-end and presented to the user to validate them.

The resulting model along with its newly created cap faces is then displayed to the user on the web interface so that its correctness can be validated. At this point, the user may also save the model so that it can later be loaded from the server.

4.2.2 Meshing

After the model is submitted, an additional meshing step is required. This ensures that the model geometry meets the requirements of the flow solver. This step uses the TetGen mesh generator tool via SimVascular to generate a tetrahedral mesh suitable for numerical methods such as flow simulation.

This step requires the user to specify the tetrahedron edge size - the smaller the edge, the finer the mesh and vice versa. This parameter directly determines the density of the subsequent simulation data points and the time needed to create the mesh and perform the simulation and is therefore very important to select sensibly. The meshing step produces the files needed for the next step, the simulation.

4.2.3 Simulation and visualization

The simulation step again prompts the user to input simulation parameters such as the inlet and outlet mappings, inlet flow rates through time, outlet resistances, number of time steps, density of result time steps (time steps where results are gathered), flow parameters and others. The mesh is displayed in 3D to the user and cap surfaces are annotated to help with their identification. The parameters all have a default value set to enable the option of a simulation with minimal user interaction.

It uses three solvers in total: pre-solver (tasked with creating the boundary conditions and creating the files required for the flow solver), flow solver (tasked with simulating the flow by iteratively searching for solutions) and post-solver which parses the simulation data and stores the resulting data in one or more files in the VTK format.

This data is parsed again and compressed to be ready for network transfer to the web interface where it is decompressed and visualized. These results can also be saved on the server so that they can be visualized later on. The results are typically stored in several time steps selected by a slider. Various visualization options for displaying different quantities are then set via a side panel, as discussed in detail in section 4.4.2.

4.3 Back-end implementation

The application consists of two parts. The back-end runs on a server and performs the meshing operations and the simulation itself. The front-end is web based and runs on any modern web browser with no additional prerequisites.

The main back-end workflow is directed from a Node.js server application which exposes a set of endpoints called by the front-end for model management, mesh creation, simulation and results management. It manages the required files and executes the corresponding processes via bash scripts.

The back-end workflow begins when an uploaded model is received from

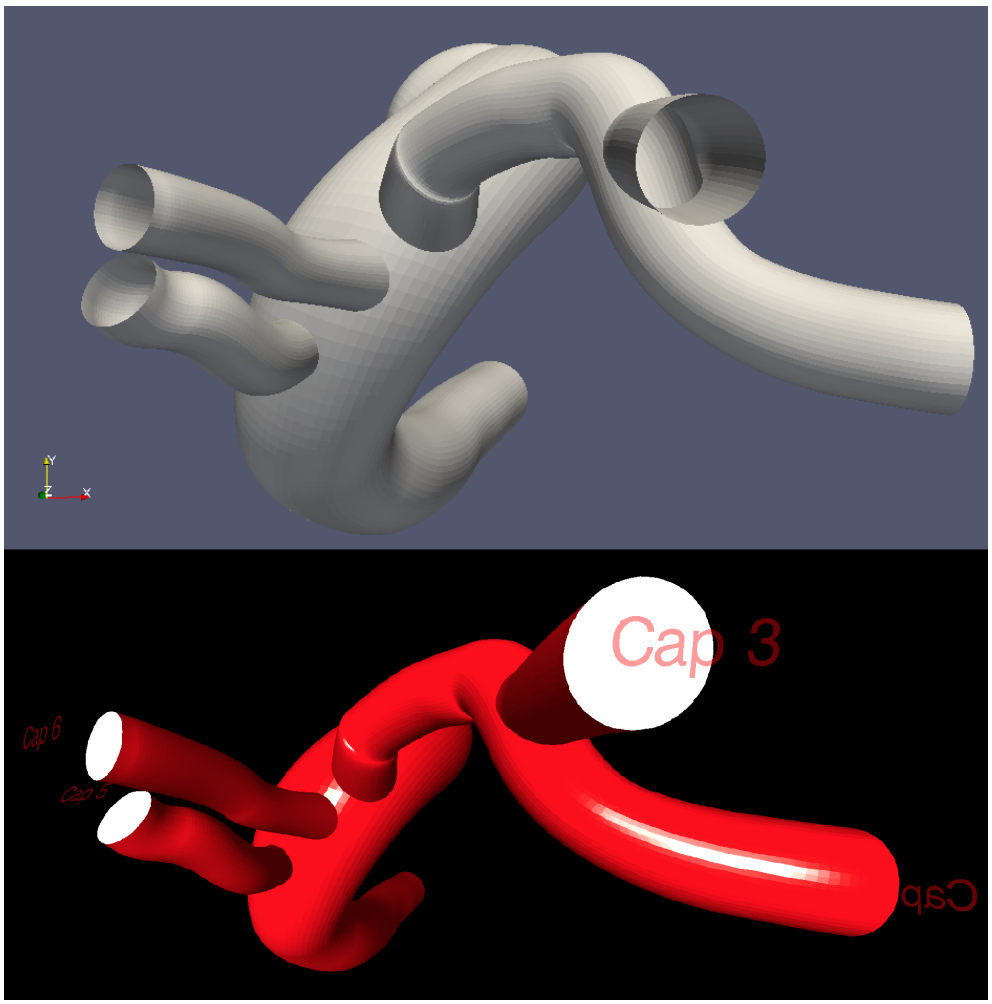


Figure 4.2: A comparison of an input Wavefront .obj model (top, rendered in ParaView) and the same model with cap holes filled with surfaces and annotated, rendered in our web application (bottom).

the user. This is a Waveform .obj file with open caps that first needs to be converted to a VTK polydata format where every model face is annotated, including the blood vessel walls and cap faces. A Python program is used to do the initial conversion, center the mesh, identify the cap holes and construct the cap surfaces. The original model and the model with cap surfaces filled are visible in figure 4.2. An identifier is assigned to every created cap surface and used for the remainder of the meshing process. The cap areas are also calculated during this step and the cap surfaces are sorted accordingly. A cap radius is estimated from its surface area for every cap surface, the purpose of which is to give the user a starting point in selecting the appropriate meshing tetrahedron edge size. After this, the cap surfaces are separately converted back to .obj format to be displayed on the front-end.

4.3.1 Meshing

The next step begins when the user provides the meshing parameters and the tetrahedral mesh needs to be created. The most important of these parameters is mesh tetrahedron edge size. A rule of thumb is that the mesh tetrahedron edge size should be at most one half the cap radius of the smallest cap to make sure the important geometry details are captured, but this is dependent on the general model geometry. If there are narrow *"choke points"* or other small perturbations the meshing process might not find a valid solution with the set edge size and can fail. This is why the initial estimate is only a starting point and further corrections might be required. To be able to use SimVascular's meshing commands, a separate script needs to be created first. The script is created dynamically for every new meshing process and contains the parameters needed by the TetGen mesh generator. A SimVascular TCL-shell wrapper is then started and the necessary shell commands are entered to complete the meshing, including writing the mesh to the disk for subsequent use in the simulation. The mesh (and cap surfaces) are also again separately converted to .obj files so that the front-end can

display the newly generated mesh. The meshing progress is printed to the console, parsed by the server application and continuously sent to the front end to inform the user on the progress.

4.3.2 Simulation

The simulation also requires a set of user-provided parameters described in section 4.2. The necessary files containing flow rate information through time for the inlets also need to be uploaded as a prerequisite. Then, two more scripts need to be dynamically generated for the simulation. The first is a pre-solver script, generated using the simulation parameters, which specifies the boundary conditions of the model walls and cap faces. This script is used by a SimVascular pre-solver process to generate the required files for the simulation. The second script is generated from the actual simulation parameters and is used by the SimVascular solver command to perform the simulation, alongside the input boundary condition and other files generated by the pre-solver command. Some parameters are hardcoded in the script itself, such as the residual criteria (the threshold upon which the solution is said to be found and no further iterations are needed) and various other default values which are usually static. The solver can work in parallel on separate threads which can substantially accelerate the simulation speed (up to 8 times on our server hardware). Mpich2, a Message Passing Interface (MPI) implementation, is used for spawning separate solver processes. The simulation reports its current state to the console, specifying the current residual error and the current time step. Like in the meshing step, this information is parsed by the server application and continuously sent to the front end to inform the user on the progress.

4.3.3 Results gathering and compression

After the simulation is finished, the results need to be gathered into a separate file for further processing. For this, a SimVascular post-processing command is used with parameters specifying the density of gathered results (the number of time steps between a keyframe step where the results are gathered). The resulting file is in a VTK format and not yet suitable for direct parsing, so we use a Python script with VTK wrappings to access the results file's data structures and retrieve the relevant time step point data for the quantities gathered, consisting of flow velocities, flow pressures, in-plane tractions, wall shear stresses and time-derivatives of flow velocities.

This data is parsed by the compression module where it is compressed and saved to a file, which is then sent to the user's front-end. More on the compression module is discussed in chapter 3.

4.4 Front-end implementation

The front end is a web application written in the Angular.js JavaScript framework. Its role is to interface with the user, fetch the required simulation parameters, and subsequently display the simulation results. Style-wise, it follows the Med3D framework [40] developed at the Laboratory for Computer Graphics and Multimedia, Faculty of Computer and Information Science, University of Ljubljana, for rendering of volumetric medical data [41].

It consists of several modules with differing functions and responsibilities described in the following sections.

4.4.1 Architecture and user interface overview

Architecturally, the application consists of several modules.

The router maps interfaces with the back-end endpoints. It translates requests such as loading the server-side model, loading the results file, up-

Figure 4.3: The main toolbar.

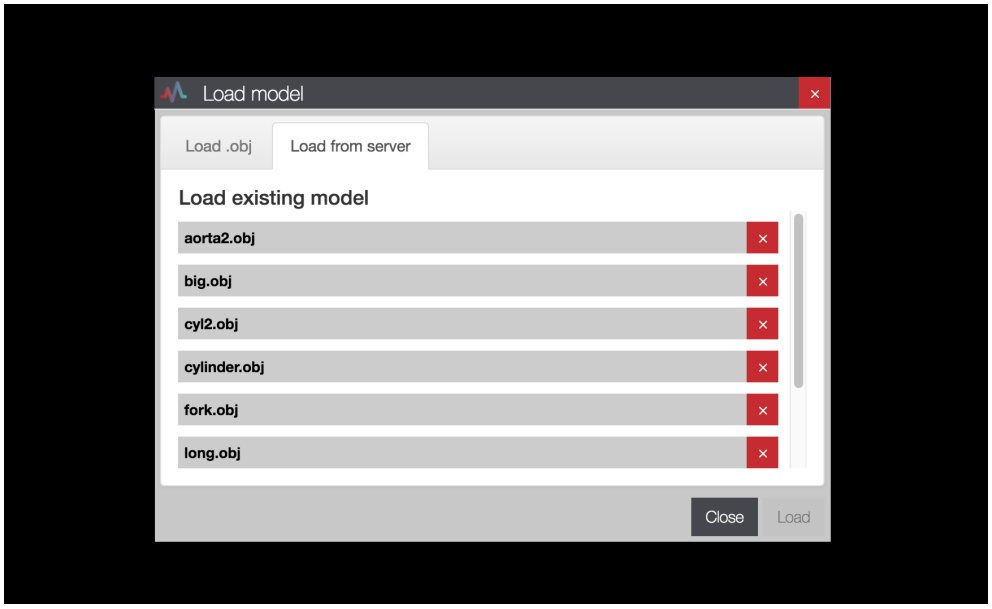


Figure 4.4: A pop-up window for loading saved models from the server.

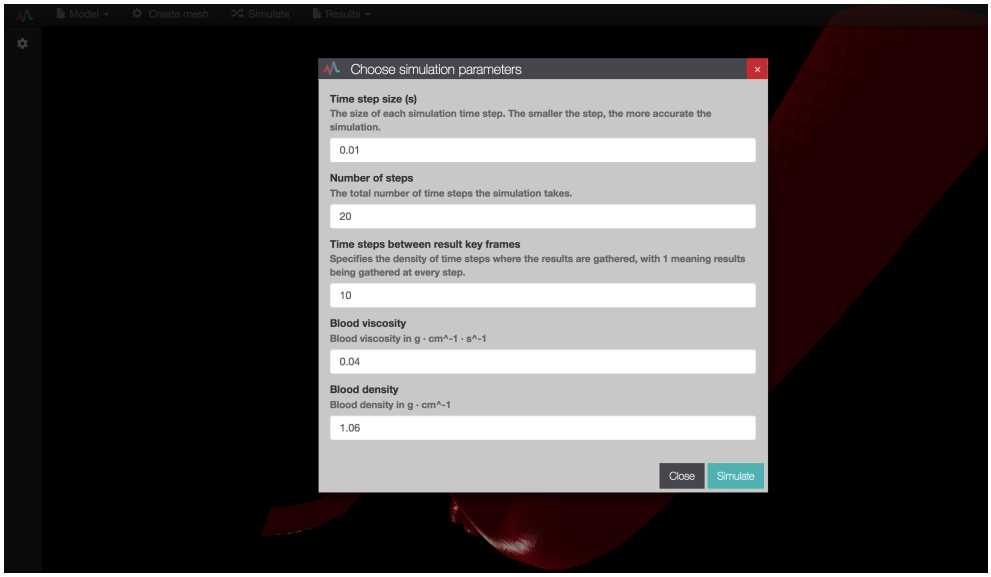


Figure 4.5: A pop-up window for simulation parameters selection.

loading the model and requesting the simulation job to network requests, waits for their execution and invokes the callback given by the calling application.

The decoder is tasked with parsing the binary results file, translating it into numeric data and decompressing this data to actual result data values. As discussed in chapter 3, the decoder also requires to create the octree and calculate the B-Spline values, as well as perform the recovery of quantized values.

The renderer generates the different visualization options (described in detail in section 4.4.2) and displays them on the canvas. It also displays the model mesh after the user submits a model and provides the ability for the user to navigate and pan the camera in 3D space.

The GUI layer presents the buttons and the inputs to the user in a intuitive way. The top toolbar (figure 4.3) guides the main workflow through the buttons for loading models, saving a loaded model, model mesh generation,

creating a simulation job, loading saved results and saving the results. The buttons are enabled or disabled depending on the state the application is in, thus guiding the user through the workflow. The button for loading the models opens a dialog box (figure 4.4) where a model can be selected either locally from the user's hard drive or loaded from the server among the previously saved models. The loaded model can be saved by clicking the corresponding button. The mesh generation button opens another dialog whose function is having the user enter the tetrahedron edge size. To aid in choosing this parameter, a list of cap sizes is also presented since this usually determines the desired edge size as well as constraining the maximum valid edge size. To give a starting point, the default edge size is set to half the radius of the smallest cap. The simulation button again opens a dialog (figure 4.5) where simulation parameters are entered - time step size, the number of time steps, the density of result time steps, blood viscosity and blood density. These values are set to defaults since the user is not expected to change most of them, especially the blood parameters - as long as the model size is set to its correct physical size in centimeters (the default unit), otherwise the blood parameters need to be updated accordingly.

The sidebar contains two panels, a panel displaying the list of inlets, outlets and their parameters (figure 4.6), and the visualization settings panel (figure 4.7). The first is populated and enabled when the user loads the model and contains the list of all caps, sorted by their area from the largest to the smallest. The user can specify whether a cap indicates an inlet (where the simulation sets a boundary condition based on the flow rate in time) or an outlet (where the simulation sets a resistance boundary condition, if there is one specified). In case of inlets, their flow needs to be specified which is done via a separate dialog in which the user can set a constant rate flow (flow rate which remains constant through time) or upload a flow file specifying flow rates through time. In case of outlets, their resistance can be set, though their default is set to 0 (no resistance boundary condition applied).

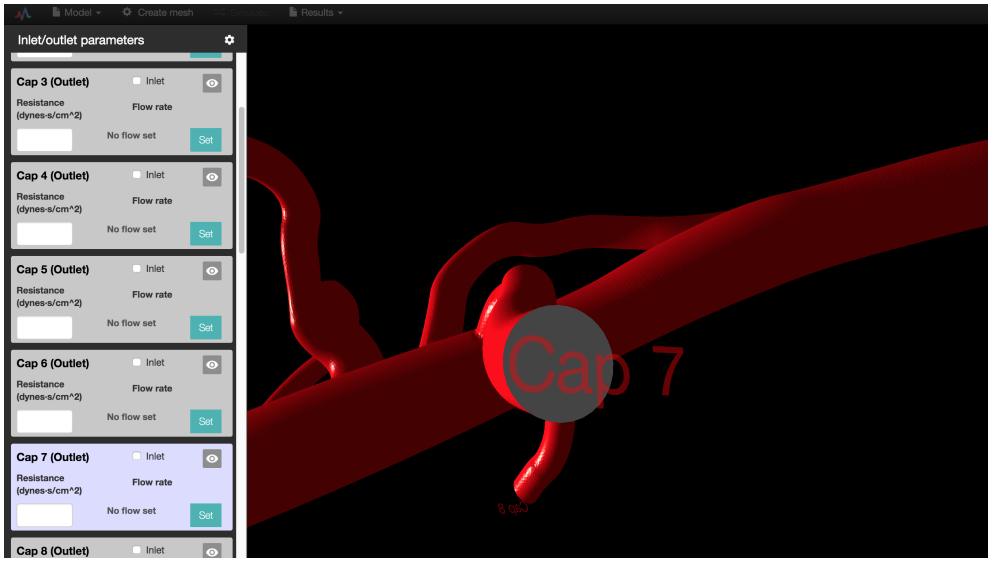


Figure 4.6: A scrollable sidebar for configuring inlet and outlet parameters for a mesh.

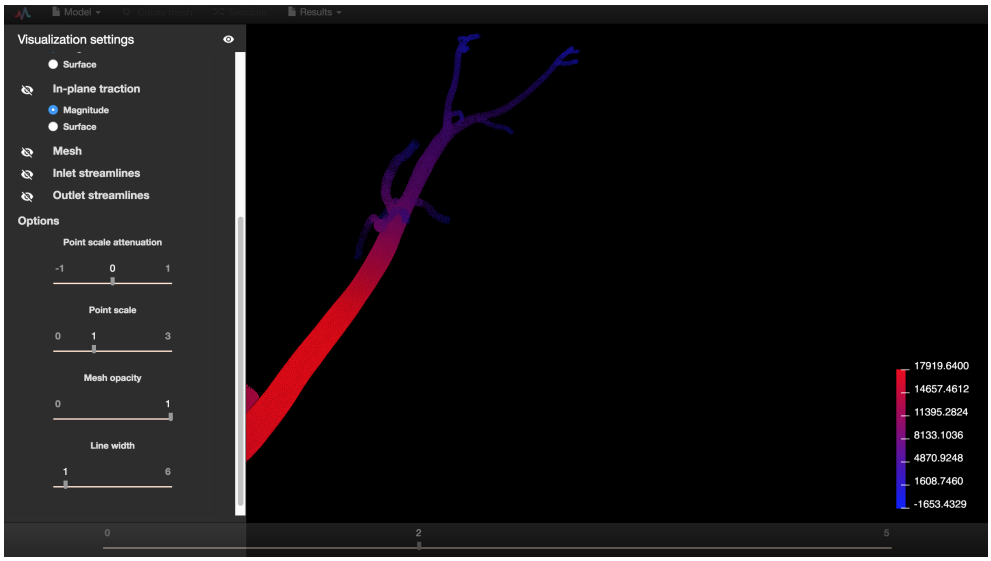


Figure 4.7: A scrollable sidebar for selecting results visualization options.

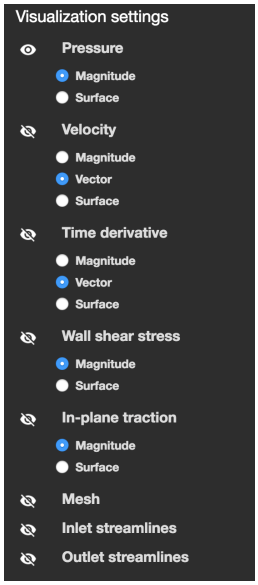


Figure 4.8: The available visualization options for different quantities.

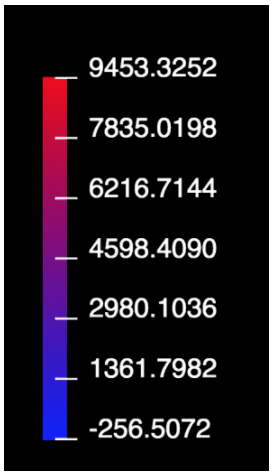


Figure 4.9: A legend showing the data range of currently displayed quantities.

By default, the first and largest of caps is set as an inlet and the rest are set as outlets, since this is usually the configuration in practice, where the larger cap is assumed to be the aorta inflow, and all others are individual branch outflows. The default flow rate is set to an constant rate inflow of 100 cc/sec. The flow could also indicate an outflow, in which case the flow rate sign is reversed. By convention, a negative flow rate indicates an inflow, which is shown as a note on top of the flow selection dialog so as to not be confusing to the user.

The visualization settings sidebar panel contains options for different visualizations. These include mesh, points, vector, streamlines and surface display as well as different options such as mesh opacity, point scale and attenuation factors and line width. These visualizations can be toggled on and off, as shown in figure 4.8, and several can be shown at the same time, as long as they are not mutually exclusive. A legend is also shown at the bottom

right corner, as shown in figure 4.9, consisting of a color gradient corresponding to the visualized value colors and a set number of tick-marks indicating the values at those marks. The legend is updated for each displayed quantity and time step.

At the bottom of the screen is a loading bar shown when the application is awaiting the response from the back-end, such as when waiting for the meshing or simulation jobs to finish, or when the results are being decompressed. In case of a running simulation or meshing job, it also displays the status the simulation job is in, such as the current time step or the results gathering phase. Also at the bottom of the screen is a slider, displayed when the results are loaded and is used to change the currently displayed time-step. The user can use it to quickly cycle through the time steps and see the flow progression through time.

When a model is loaded, the mesh is displayed on the renderer canvas, as are its caps, colored in a different color. The caps are highlighted when the user hovers the mouse cursor over them, which is done by sending a ray from the camera position and checking for intersections. A text element displaying the cap index is also rendered in 3D near the cap surface. When a cap is highlighted, the corresponding list element in the sidebar panel is also highlighted and vice versa. The list element in the sidebar also contains a button that, when clicked, moves the camera to the corresponding cap object on the mesh. This way, the user has a good understanding which cap in the sidebar list corresponds to which cap surface on the 3D canvas.

4.4.2 Visualization

The following section presents the different developed visualization types available to the user to visualize the simulated results.

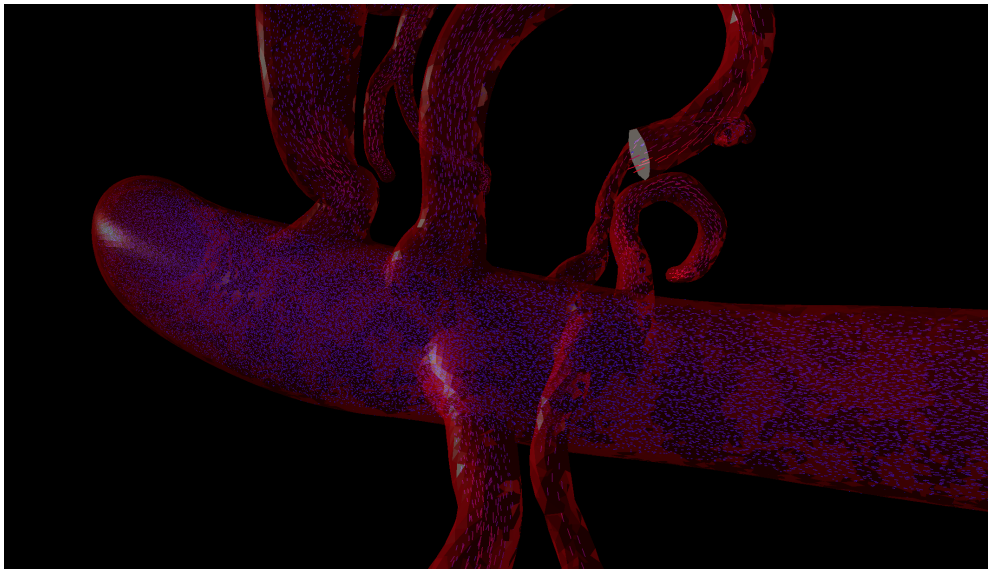


Figure 4.10: A semi-transparent mesh with velocity vectors shown inside.

Mesh

The model mesh and its caps are displayed on the renderer canvas after the user loads the model, and are updated with the tetrahedral surface mesh after the meshing process is done. The tetrahedral mesh can be displayed alongside the result data, once it is available. A mesh opacity slider controls the mesh and caps' material opacity so that the simulation data inside the mesh can be viewed simultaneously, as seen in figure 4.10.

Vector display

For velocities, the most natural way to display is by vectors in 3D space, e.g. lines along the vector direction whose lengths correspond to vector magnitudes. Because the number of these lines can be in the hundreds of thousands, they are stored in an efficient Three.js (section 4.1.2) "BufferGeometry" object, which stores vertex positions, vertex (line) colors etc. in buffers, making it faster to transfer the data to GPU. Each line is also assigned its color in-

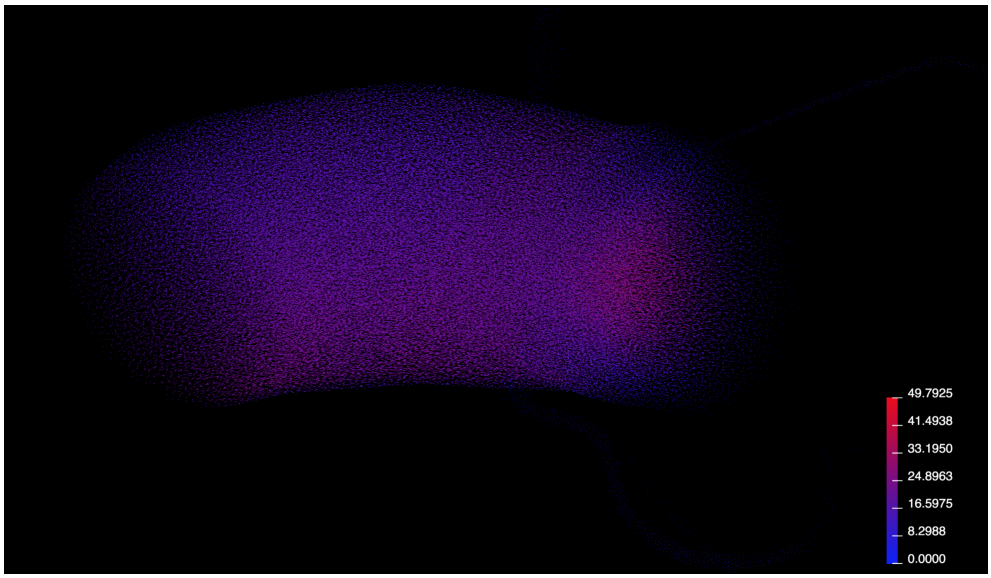


Figure 4.11: Velocity vectors inside the main coronary artery.

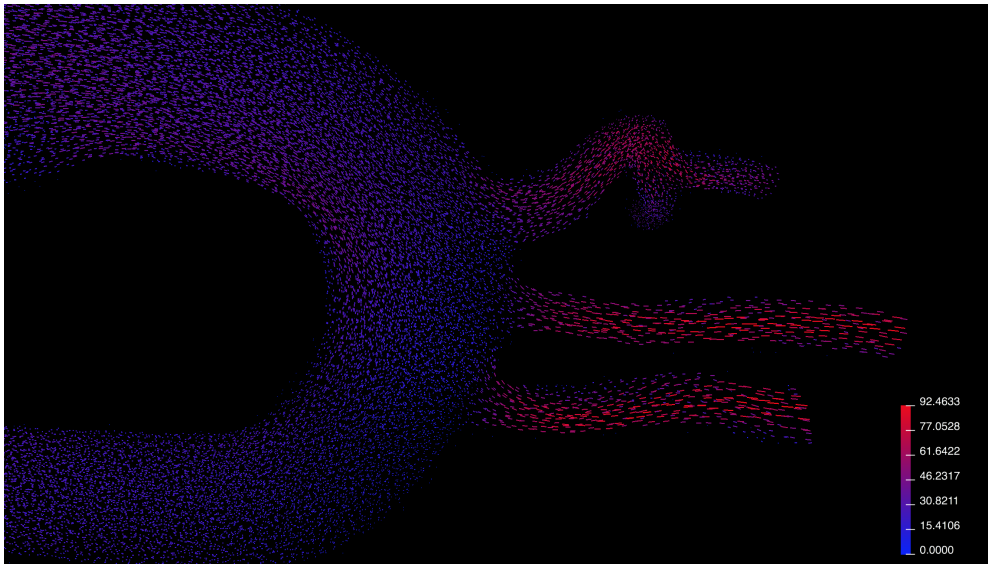


Figure 4.12: Velocity vectors of arterial outlets on a model of aorta.

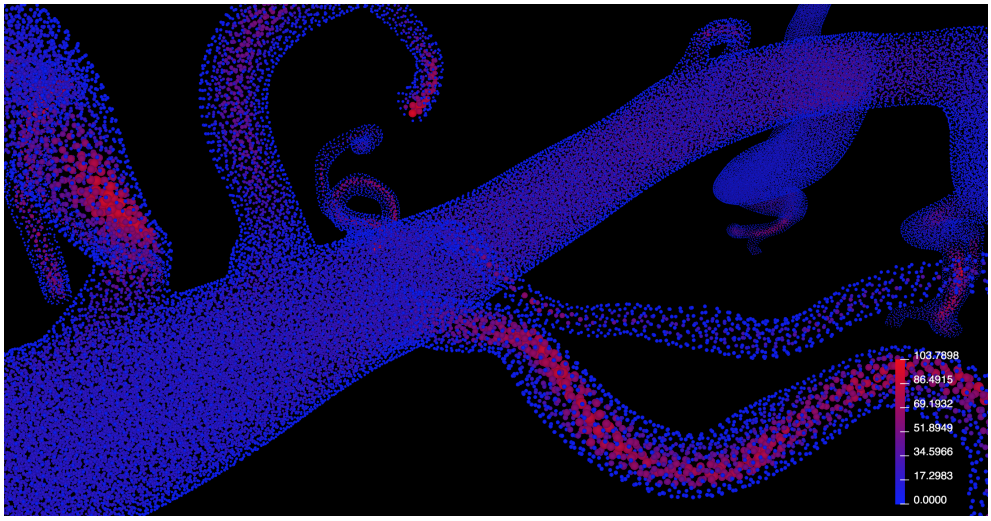


Figure 4.13: Velocity magnitudes displayed as scalar points.

dicating the vector magnitude for easier distinction by the user. The color is linearly mapped from the ratio between the vector magnitude and the global value range to an RGB color value by the formula

$$(ratio * 255, 0, (1 - ratio) * 255). \tag{4.1}$$

When the results for a certain time step are requested, the velocity vector lines are created and cached so that any subsequent request does not need additional geometry construction. This visualization type is very informative in conveying to the user the flow direction and its local velocity. More slowly flowing regions will then be colored in blue hues and with shorter lines, while faster regions will stand out in red and longer lines. Two examples of data visualized as velocity vectors are shown in figures 4.11 and 4.12.

Scalar points

This visualization type displays the data points as spheres whose size correspond to the scalar values of the selected quantity. This type can be used

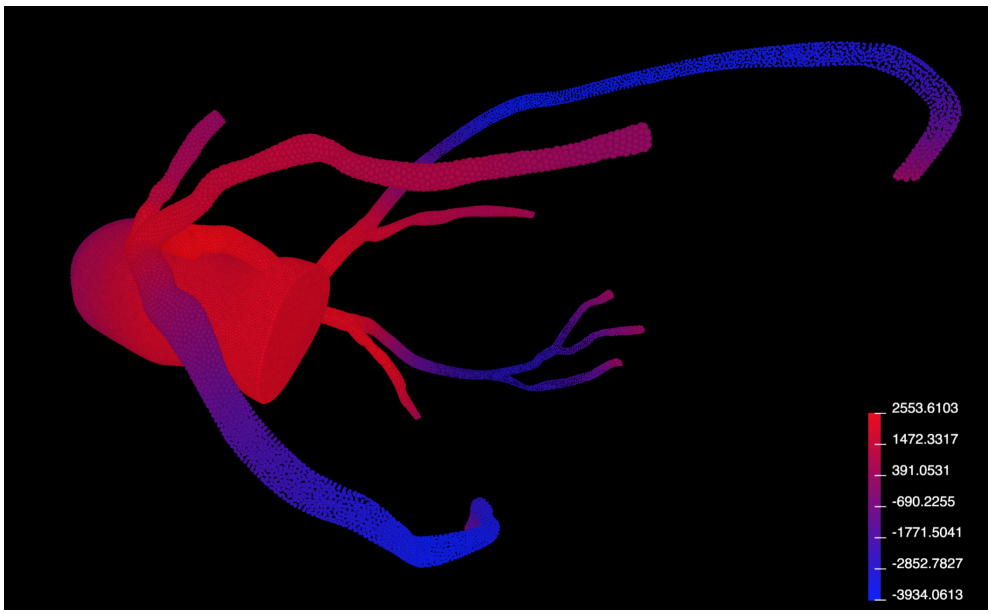


Figure 4.14: Pressures on a model of a coronary artery displayed as scalar points.

to display pressures as well as velocity magnitudes, in case of which these need to be calculated first and are also cached. The spheres are also colored according to their value, similar to the vector line display - by mapping the ratio of their value to the global value range. These points also need to be constructed efficiently due to the possibility of hundreds of thousands of data points. For this reason, WebGL vertex and fragment shaders were written to render the individual points - their size, color and translation in space. The points are created only once, since their positions (translations) do not change throughout the time steps. Their sizes and colors can then be changed dynamically on each time step by changing the so called uniforms array the shaders have access to. One of these uniforms is a point scale factor which controls the additional scaling factor and another is a point attenuation factor which controls how attenuated the sizes of the points with larger scalar values are. The larger the attenuation factor, the more pronounced the larger points and the less pronounced the smaller points. These two factors can be controlled by two sliders which gives the user the ability to fine tune the points display to the user's needs. For performance purposes, the spheres are not 3D objects but instead shaded 2D sprites to give the impression of depth. This, combined with the use of WebGL shaders, enables very efficient rendering even with hundreds of thousands of points. Point scales, attenuation factors or colors are also changed in real time without any stutter, making for a smooth real-time user experience.

This visualization type effectively presents the pressure values and velocity magnitudes. High-pressure or high-velocity regions stand out among low-pressure or low-velocity regions because the first are larger and the later are scaled down, making it possible to see "through" the low-pressure or low-velocity regions. Two examples of scalar values of velocity magnitudes and pressures are shown in figures 4.13 and 4.14.

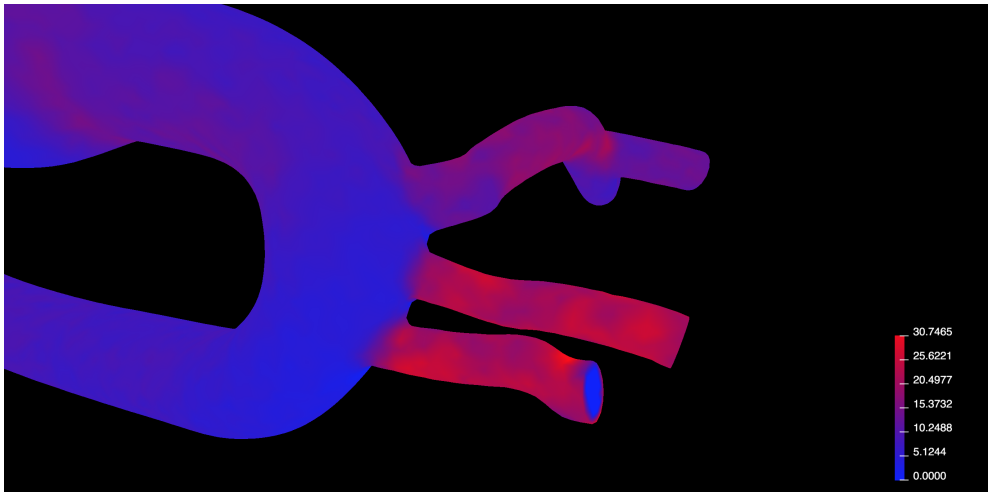


Figure 4.15: The surface display showing shear stresses along vessel walls.

Surface

This type is useful because we are often interested in quantity values close to or on the vessel walls. This is especially true for quantities that only make sense on vessel walls such as in-plane tractions and wall shear stresses, but also for pressures if we are interested in pressure stresses on the walls. Quantities with zero-velocity boundary conditions on wall surfaces will still display useful information on inlet and outlet surfaces, where 2D flow profiles can be clearly seen.

This visualization type uses a mesh model to which a texture map is created and assigned. This map is composed of color values for each of the data points lying on wall and cap surfaces. Since the mesh is the same tetrahedral mesh used for the simulation, its vertices correspond to data points lying on the vessel surface. The triangles are then colored according to a linear interpolation between the three vertex color values to achieve smooth color transitions. An example of wall shear stresses displayed with this type is seen in figure 4.15.

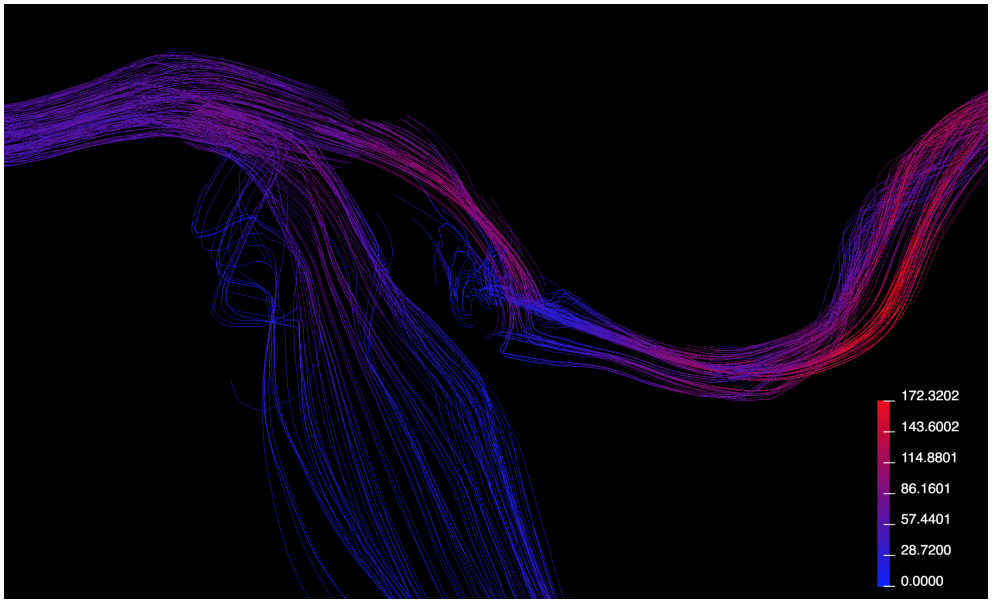


Figure 4.16: Streamlines showing turbulent flow patterns.

Streamlines

Another useful visualization option is the display of velocity streamlines. Velocity vectors convey the local velocity fields, but streamlines convey their relationships in space and show the user a continuous flow direction throughout the mesh. In flow vector fields, streamlines are all curves tangent to the flow velocity direction. A point on a streamline shows the direction of the flow at that point. A basic streamline does not hold any information about the flow's velocity magnitude at a certain point, but we can include this information by coloring the sections accordingly, in the same way as in previous methods (4.1).

The streamlines are generated using the Euler method: given the current value y_n and using a step size h, we move in the direction of flow velocity at the current position $f(pos_n)$ and repeat the procedure until flow velocity at

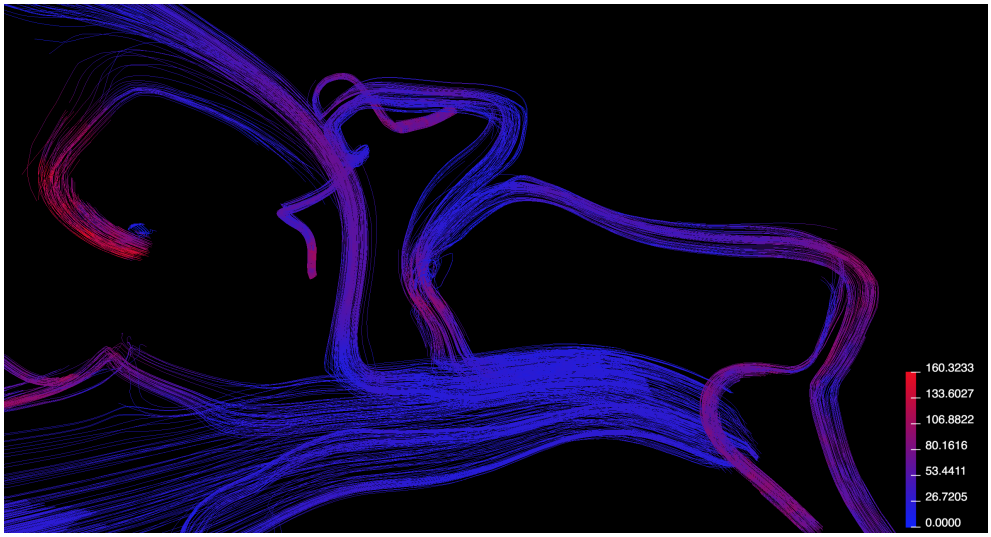


Figure 4.17: Streamlines showing mostly laminar flow through multiple outlets.

the current position reaches zero:

$$y_{n+1} = y_n + hf(pos_n). \tag{4.2}$$

The step size h is kept constant and was determined experimentally where we balanced performance and quality - larger step sizes mean rougher and less accurate streamlines but faster streamline generation. We also limit the maximum number of steps to a constant value since there are instances where a streamline can get "trapped" in a loop, especially in the areas of vortices.

The streamline starting points are randomly sampled on cap surfaces by approximating the surface as a disk with radius rad, sampling random points along its local x and y axis from the estimated cap center and only taking those within 90 % of the estimated cap radius: $\sqrt{x^2 + y^2} < 0.9\ rad$. The 90 % figure is chosen so that the streamlines are not too close to vessel walls to prevent them ending prematurely (since velocity field close to the edges is usually lower). This way, we create a given number of starting points on every cap surface so we can cover the most interesting possibilities with only

a finite number of streamlines (currently set to 150 separate streamlines per inlet or outlet surface).

Since we only have discretely sampled data points, we don't know the flow velocity in any given position. Therefore, given a position P we would like to calculate the velocity approximation in, we take a neighborhood region of data points close to it P_i and take a weighted sum of their velocity directions V_i where a data point's contribution is inversely proportional to its squared distance to our position:

$$\frac{\sum_{i=0}^{n} \frac{V_i}{|P-P_i|^2}}{\sum_{i=0}^{n} |P-P_i|^2}. \tag{4.3}$$

Since selecting this neighborhood can be computationally intensive if done naively, we first create a K-D tree of point positions. A single neighborhood query is then efficient and has an average time complexity of $O(log\ n)$. Nevertheless, generating a large number of streamlines is still computationally consuming, so the streamlines are only created once and cached for every selected time step.

Additional tweaks are required to ensure desirable display of streamlines, such as stopping the streamline once it reaches the areas near the wall where the velocity magnitudes tend to be very small and, since our compression method only has a limited accuracy, we cannot rely on their correctness. The algorithm also stops when the maximum number of neighboring points in a K-D tree falls below a set number (streamline reaching vessel walls) or when all the neighboring points' magnitudes are below a set threshold (streamline entering a slow velocity field where our limited accuracy is not suitable for further flow direction estimation).

Two types of streamlines are defined: inlet and outlet streamlines. Inlet streamlines begin at inlet caps and follow the flow in its direction. Outlet streamlines start at outlet caps and follow the flow in its reversed direction. Inlet streamlines should thus start at one of the inlets and end either at one of the outlets, or somewhere in between, since due to the numerical inaccuracies

of the method not all solutions follow the flow the entire way. The reverse holds for outlet streamlines. Disregarding a streamline's beginning and its end, they are functionally equivalent. By combining these two types, we can have a fuller display of streamlines since we can only sample a finite amount of streamlines at every cap and it is therefore not guaranteed that, for example, a certain inlet's streamlines will cover all the outlets. These two types can be toggled on or off in the visualization sidebar panel. Two examples of different flow patterns shown by streamlines are seen in figures 4.16 and 4.17.

Chapter 5

Results and evaluation

The application was deployed on a Ubuntu server with 8-core, 16-thread Intel Xeon CPU enabling fast simulations (the tested scenarios all completed within a few minutes) and was also tested on a MacOS system. To evaluate the performance of our application, we tested the results on seven different blood-vessel models, most of which were modeled after real volumetric scans in SimVascular or other modeling and segmentation tools. We needed to validate both the proper input parameter parsing - the inlet-outlet mappings, flow rates and resistances - as well as how the visualized results (after a lossy compression) compared with an external tool (ParaView). Another thing we tested was rendering performance since real-time 3D interaction is crucial for informative visualizations. While the web interface was intended for larger displays, the visualizations were rendered with smooth framerates even on mobile phones, including the largest of scenarios tested in table 5.1, which is an important advantage of our web-based approach with server-side computational offloading.

Table 5.1 shows the model set used and their corresponding complexity in terms of the number of cap surfaces and the number of volumetric tetrahedral mesh points after the model was meshed with a set edge size (relative to the model). The models as displayed on our web application renderer are shown

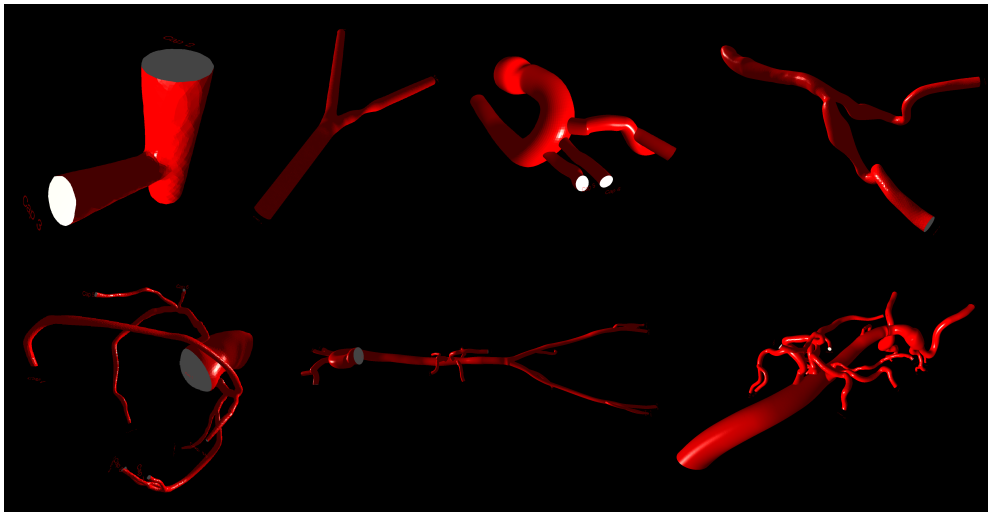


Figure 5.1: The models used in our evaluations, as rendered in the web application, listed in table 5.1, from top left to bottom right.

in figure 5.1. The table also shows the total number of data points in 6 and 16-step simulation scenarios. Apart from the first and the simplest model, a double-cylinder, the models were modeled after real MRI scans and their sizes correspond to real sizes of blood vessels they are modeled after, which is of importance for the subsequent simulation.

	DoubleCylinder	ForkFlat	AortaSmall	ForkSmall	Coronary	AortaBig	ForkBig
caps	3	3	6	3	11	18	20
mesh pts.	42864	50674	185447	107652	212519	132049	181411
6 st. pts.	257184	304044	1112682	645912	1275114	792294	1088466
16 st. pts.	685824	810784	2967152	1722432	3400304	2112784	2902576

Table 5.1: Model set used along with the number of caps, mesh points and total data points for 6 and 16 simulation steps

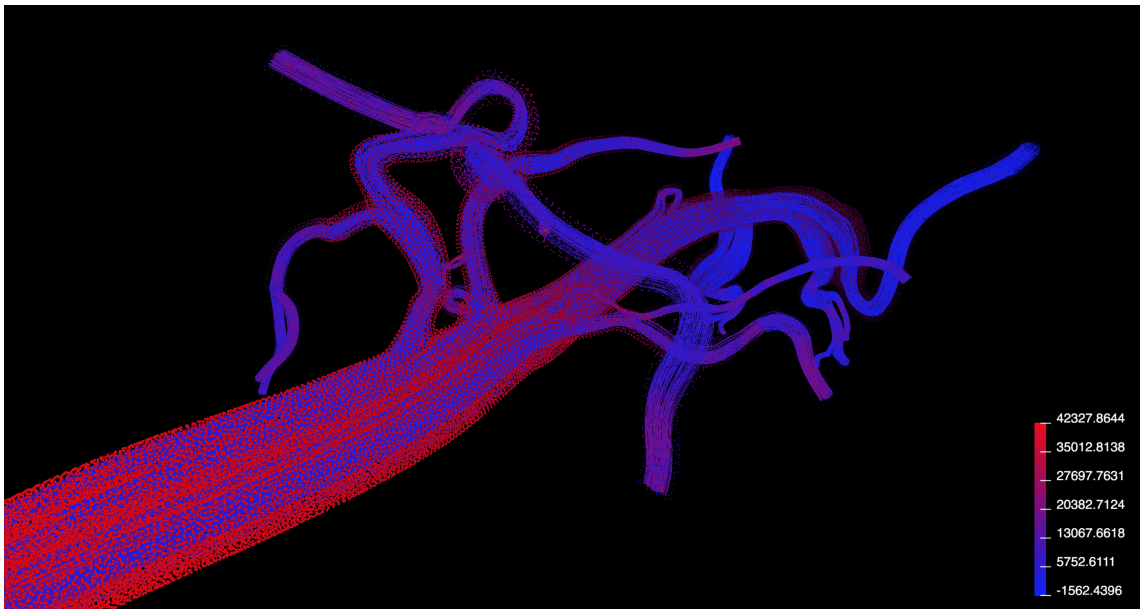


Figure 5.2: Streamlines and pressures on a model of aorta, with one inlet (left side) and 19 outlets.

5.1 Simulation results evaluation

The first part of our evaluation was ensuring the input parameters affect the simulation as intended. For this task, we tested several models using different inlet-outlet mappings and different flow rates and outlet resistances, and visually evaluated the simulation results.

Figure 5.2 shows the most typical situation of a single inlet and multiple outlets. Streamlines show the flow passing as expected through all outlets, while scalars (and the legend) show pressure, which, as expected, can be seen to smoothly decrease as the flow passes through more and more outlet branches.

We also tested different configurations of inlets and outlets. Figure 5.3 shows a scenario with two inlets merging and flowing through four outlets with zero downstream resistances set. The streamlines show the flow al-

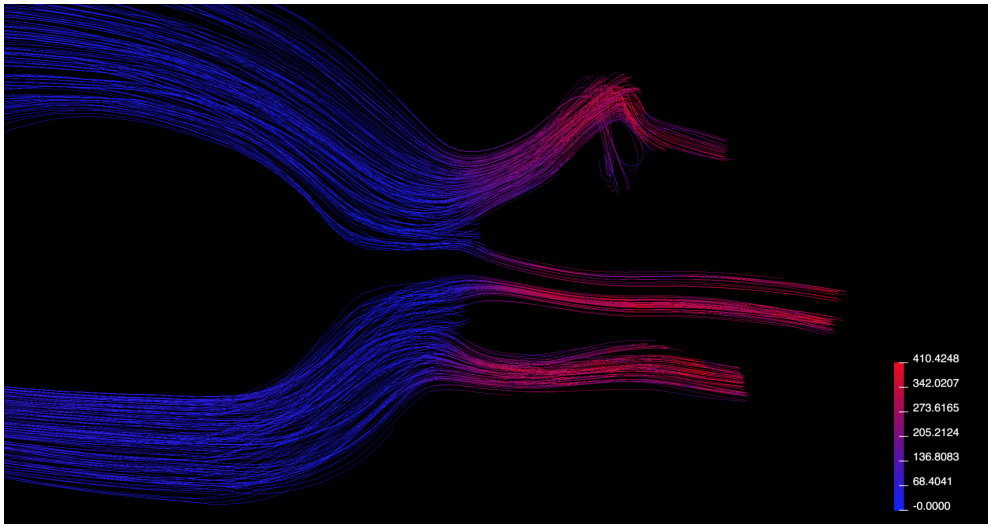


Figure 5.3: Streamlines of two inlets (left) flowing through four outlets.

most evenly splitting in two and each mostly flowing through the two closest outlets.

Next, we applied a high downstream resistance value to the bottom two outlets, shown in figure 5.4. As expected, the high resistances induce high pressures on the bottom two outlets and almost no flow passes through, with every sampled streamline passing through the top two outlets where a very high negative pressure is formed.

The configuration is inverted on figure 5.5, where the right four inlets flow through the two outlets on the left. The geometry of the mesh and four concurrent inflows cause a highly turbulent flow, as seen by the streamlines following vortex loops. The high flow rate through each of the inlets causes the flow to hit the vessel walls and deflect in vortices before gradually settling down as they pass the outlets.

A cap surface can also have an out-flowing flow rate (flowing “out” of the model) specified, in which case it acts as an outlet, and the remaining inlet surfaces do not need to specify additional flows. This is seen in figure 5.6

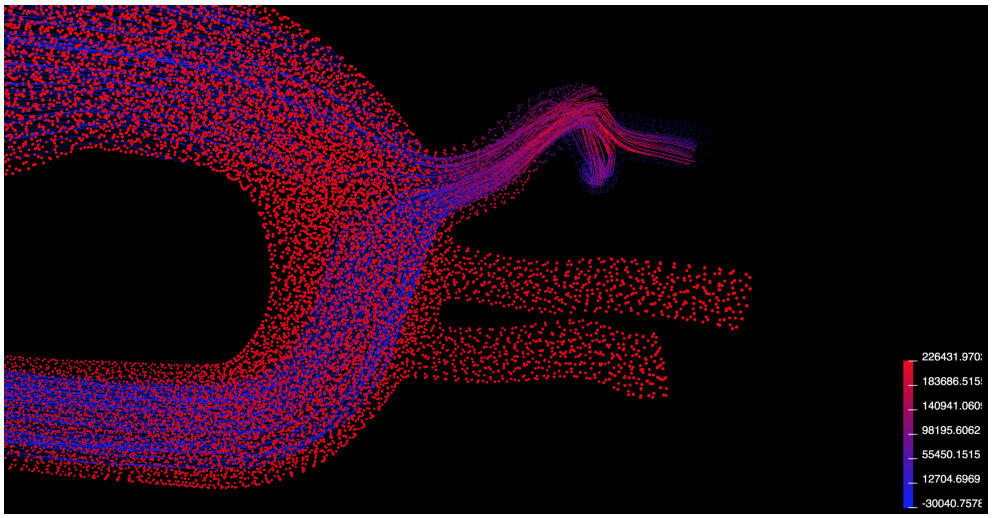


Figure 5.4: Pressures and streamlines of two inlets (left) flowing through four outlets (right) with varying downstream resistances.

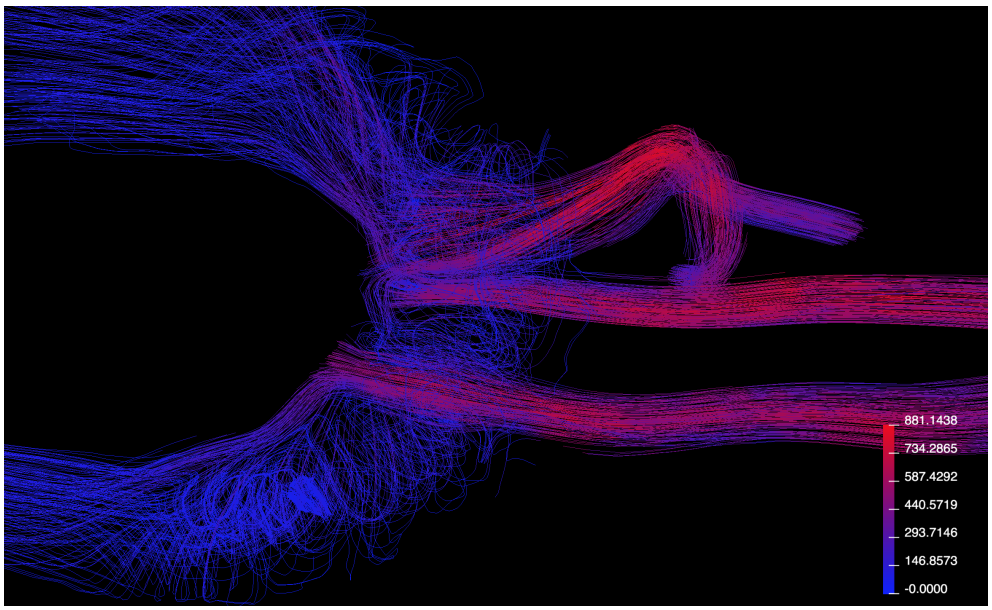


Figure 5.5: Streamlines showing turbulent flow of four inlets (right) flowing through two outlets (left).

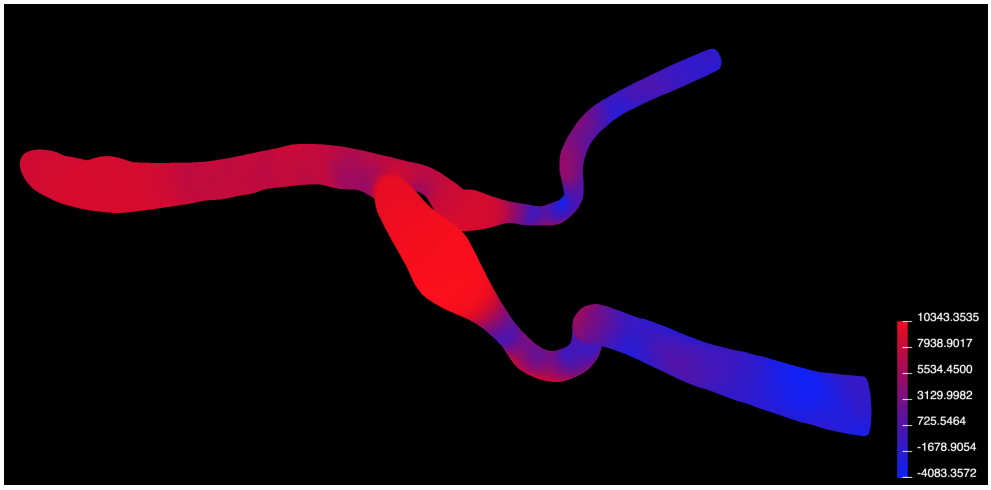


Figure 5.6: Surface pressures on a mesh with one out-flow (left) and two caps acting as inlets (right).

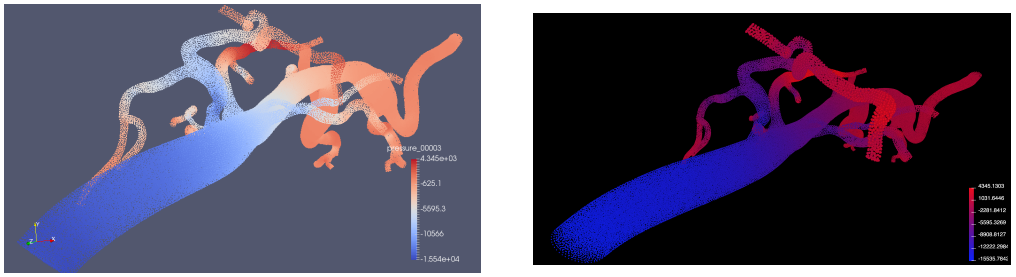
where the outflow on the right causes high negative pressure to form on the two remaining caps where the flow is "sucked" in.

We concluded that the simulation behavior was responding to input parameters as expected, as can be seen in the previous examples. The examples show the results after being compressed and rendered in our web application, but they were also evaluated while still in VTK format and displayed with ParaView to ensure their correctness. Figure 5.7 shows the same data displayed in ParaView and our web application (after being compressed and decompressed).

5.2 Compression approaches comparison

To evaluate the four compression methods - basic quantization, time-domain quantization, octree quantization and our hybrid method, we evaluated the performance on the set of models described in table 5.1.

We tested the compression of two main quantities - pressures and veloc-



(a) ParaView **(b)** Web application

Figure 5.7: Comparison of pressure visualizations in ParaView and our web application

ities - over four scenarios, dividing them in terms of compression accuracy (6-bit / 8-bit error target or 0.09 % / 0.39 % maximum average error rate, respectively) and the simulation parameters. One group of results used a steady flow over 6 consecutive time steps (with all of them captured in the results), while the other used a sinusoid-shaped pulse flow, varying through 16 consecutive time-steps. This way we attempted to produce situations best suitable for different approaches and quantities.

Charts 5.8, 5.9, 5.10, 5.11 present the comparison of the four compression methods, described in sections 3.2, 3.3, 3.4 and 3.6, in terms of the average number of bits needed to encode a single data point (each 1D component in case of 3D velocity vectors).

It shows the varying effectiveness of time-domain quantization and its sensitivity to the number of time steps and sequential data variation. This method is ineffective in encoding pressures over a short number of steps, as it is takes 36.5 % more bits per point than basic quantization in this case. Flow pressures tend to vary widely over consecutive time steps and the average data range of most data points through time is not significantly smaller than the global data range, while the number of time steps is too small to alleviate the additional bit overhead this method produces for every data point. The overhead is somewhat reduced at 16 time steps, bringing

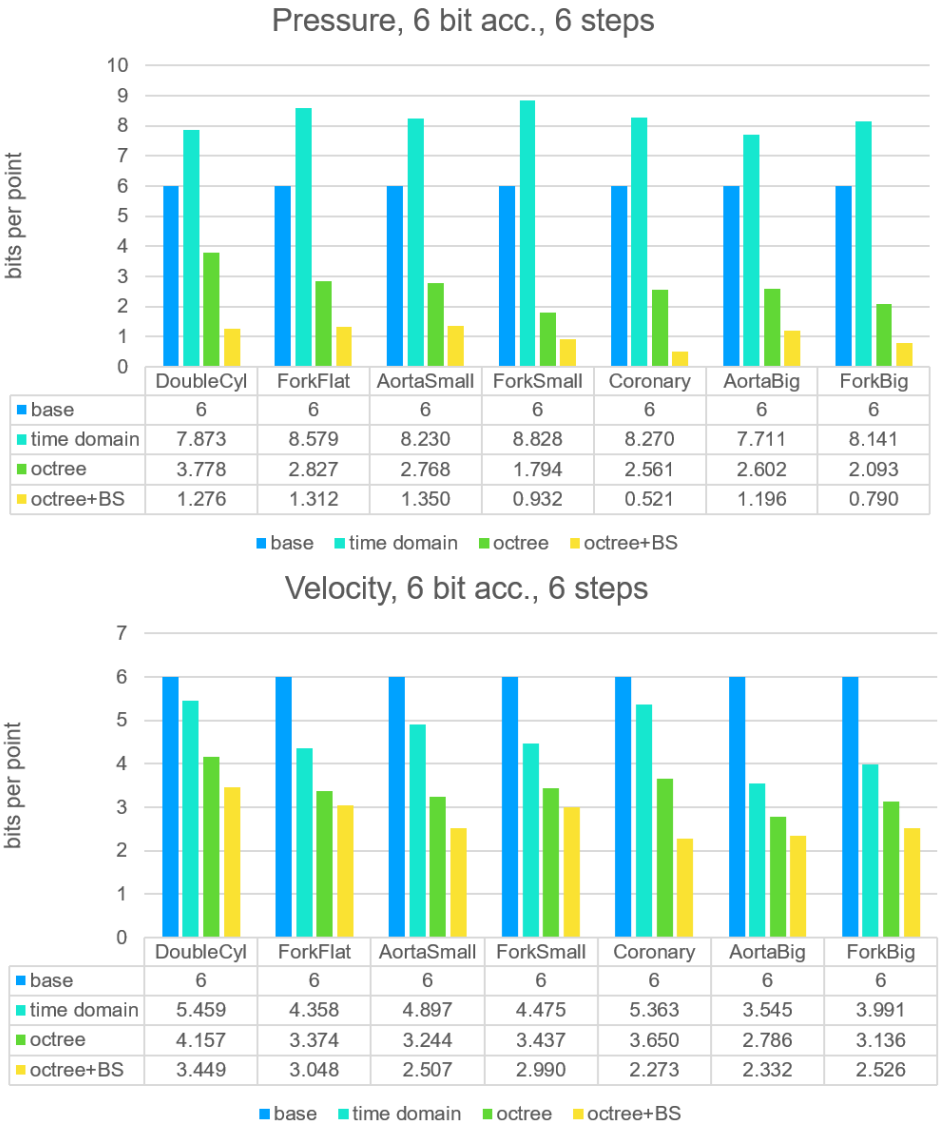


Figure 5.8: Comparison of compression methods at 6 bit accuracy target (max. avg. error of 0.39 %), 6 time steps.

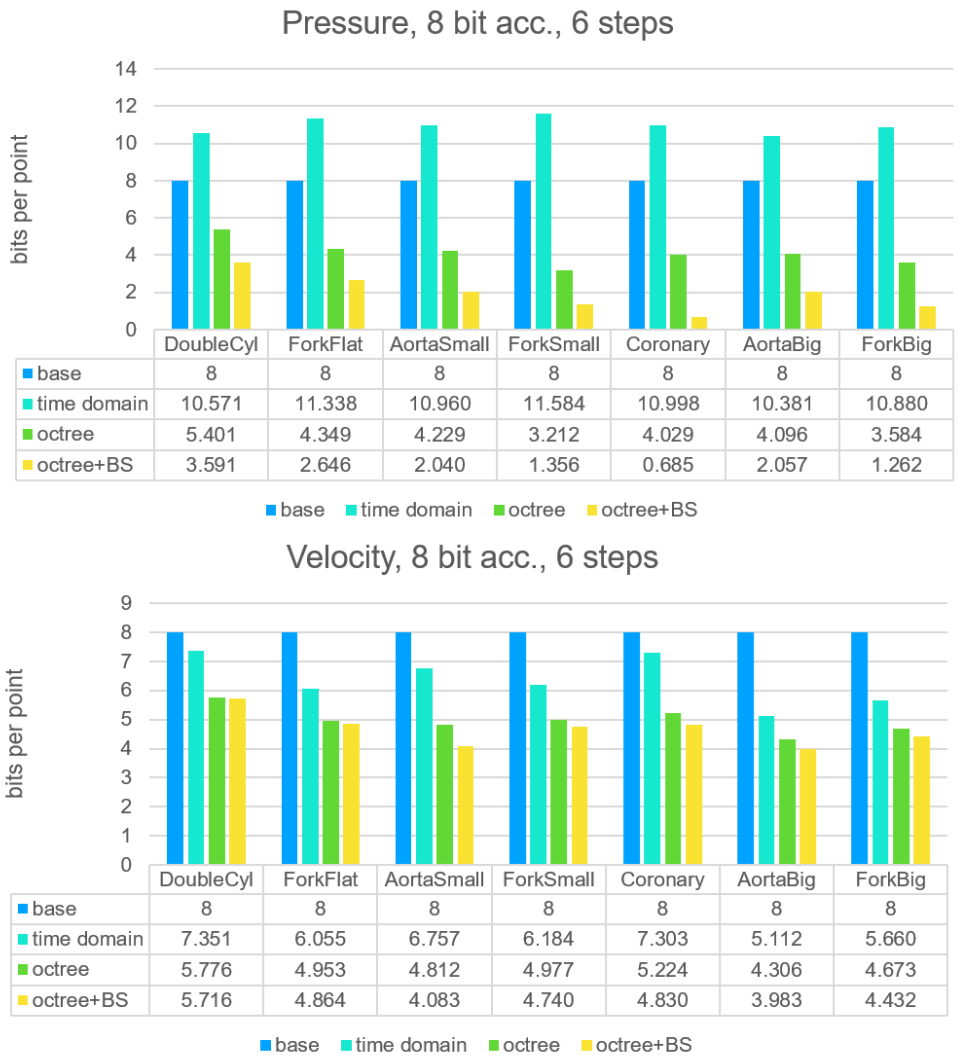


Figure 5.9: Comparison of compression methods at 8 bit accuracy target (max. avg. error of 0.09 %), 6 time steps.

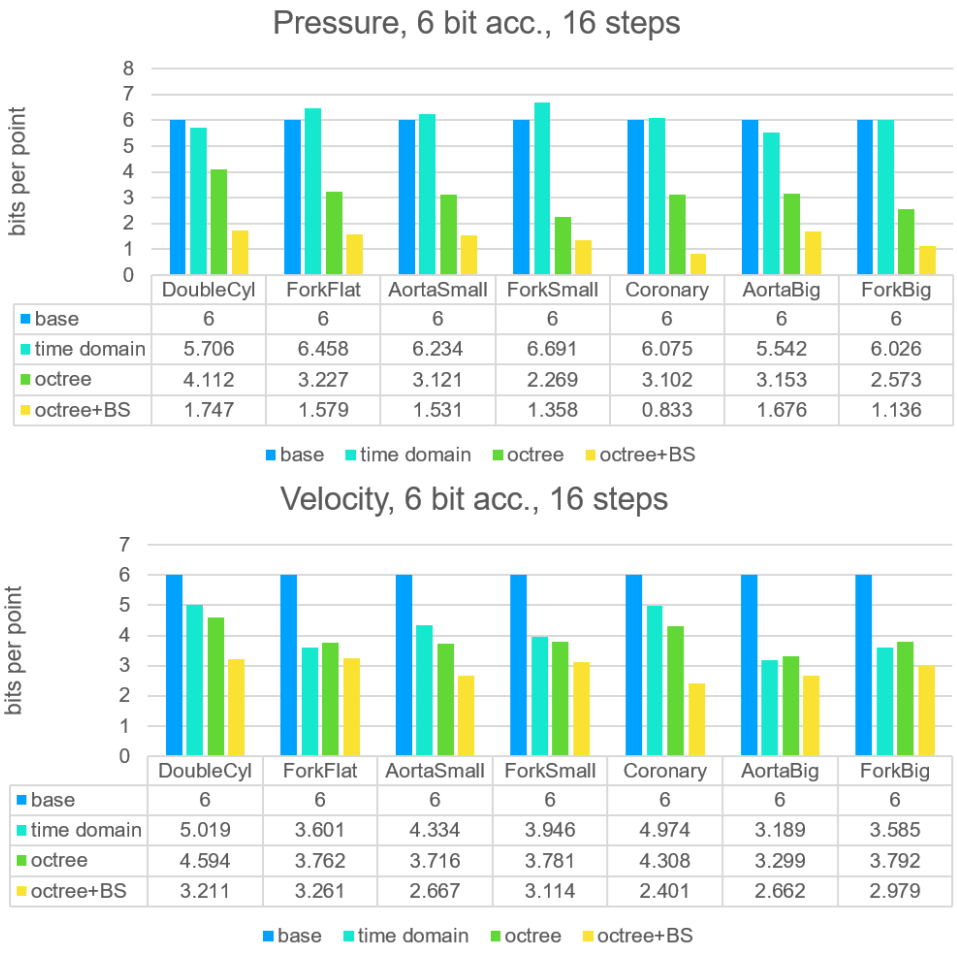


Figure 5.10: Comparison of compression methods at 6 bit accuracy target (max. avg. error of 0.39 %), 16 time steps.

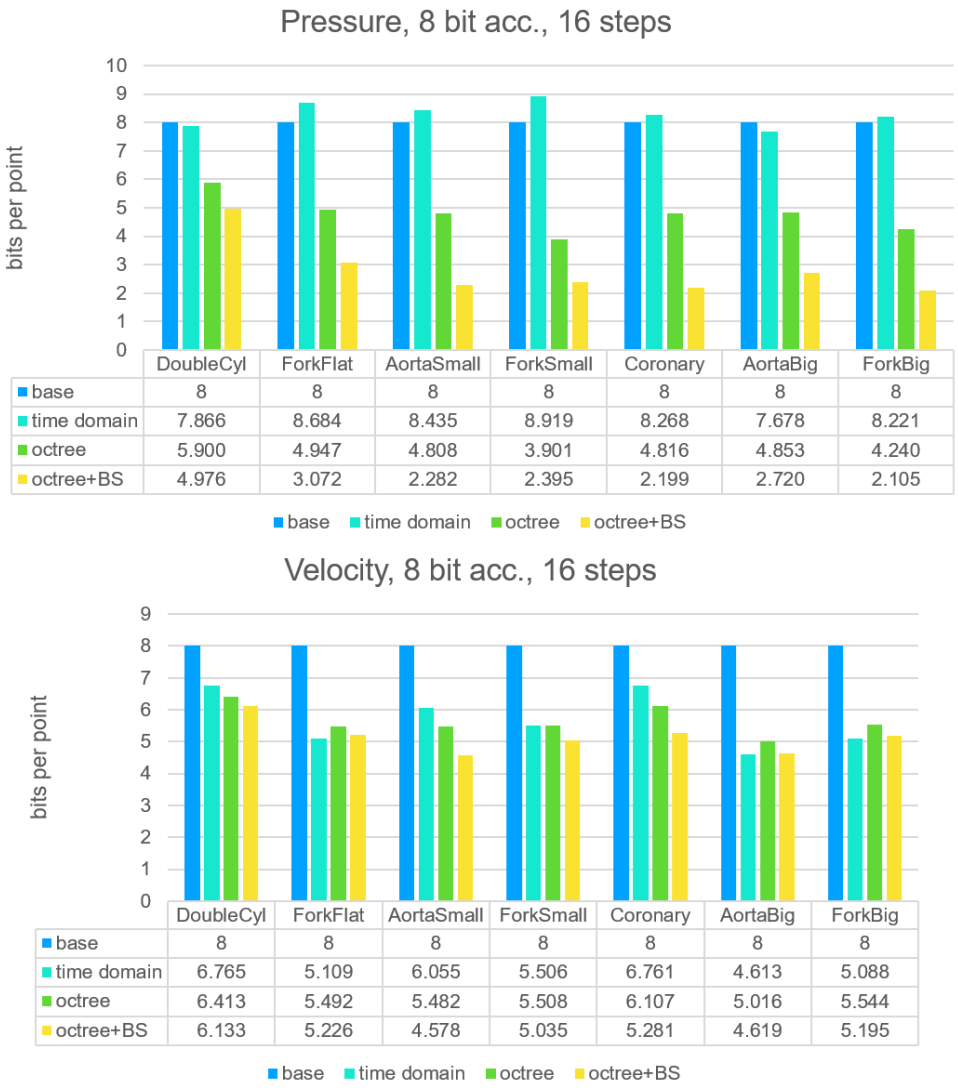


Figure 5.11: Comparison of compression methods at 8 bit accuracy target (max. avg. error of 0.09 %), 16 time steps.

this method's performance roughly on par with basic quantization. Velocities tend to vary much less and are therefore much more suitable for this method, requiring on average 28 % and 43 % fewer bits per point for 6 and 16 time steps, respectively. In certain scenarios when encoding velocities over a larger number of time steps, this method is the most efficient among the ones implemented, however in general it is only suitable for velocities in certain scenarios and not for pressures.

Quantization of octree subdivisions is generally more effective in all cases, requiring on average 40-56 % and 30-43 % fewer bits per data point for pressures and velocities, respectively, compared to the basic method. Velocity vectors are somewhat less suitable for this method because they can often take widely ranging values in local regions, requiring more bits per region. This is less of a case in pressure values, as the results imply that local subdivisions contain very small value ranges, enabling data to be encoded in 2-4 bits per point.

Finally, a hybrid method using a mix of B-Spline regression and quantization in octree subdivisions is always more effective than the method only using octree quantization, and only in a single instance minimally less efficient than time-domain quantization. The amount of improvement over the method using just octree quantization is larger in case of pressure values than it is in case of velocity vectors, ranging from 40 % to 55 % and 6 % to 25 % fewer bits per point on average for pressures and velocity vectors, respectively. It is also very sensitive to the initial error rate target, being much more efficient on larger maximum average error rates (e.g. 6 bit target or maximum average error of 0.39 %), especially in case of velocity vectors.

Velocity vectors encoded under a stricter accuracy constraint (8 bit target or maximum average error of 0.09 %) are the least efficient scenario, providing 6 % to 9 % fewer bits per point compared to just using octree quantization. This is due to larger numbers of "outliers" skewing the smoothness of the surface and making it harder for a regressed B-Spline volume to fit the data,

making the error rate unsuitable and thus requiring more control points to meet the error rate criterion. If the error criterion is relaxed, the iteration process can find a suitable B-Spline model in a significantly larger portion of regions and the thus effectiveness of the method is larger. This is also the reason why this method is more efficient when encoding pressure values, since these values usually have a much lower number of outliers and their transitions are much “softer”.

Chapter 6

Conclusions

Simulation of cardiovascular systems and informative display of results is an interesting field with great past record and even greater future promises in providing alternative means for cardiovascular disease study and surgery planning. In this work we developed an application for accepting cardiovascular models, simulating blood flow inside them and visualizing the results in different ways. We exposed the functionality through a web application with a simple and intuitive GUI to enable fast workflow iterations and the ability to load previously saved results and models. We connected the web application to the back-end application where the computationally intensive parts of the process are performed. We implemented different ways of visualizing the results through an interactive 3D canvas. We also implemented a method for result data compression to minimize the amount of storage needed and maximize network file transfer speeds.

Our evaluations showed successful simulations and visualizations in line with provided parameters and expectations. The model conversion, meshing and simulation processes performed as expected and the simulation results mirrored the desired changes to the input parameters. The visualizations displayed interesting features formed by blood flow and were each able to provide an informative and complete display of data. Their performance was

good enough to enable smooth real-time interaction even on mobile devices. The developed compression method was able to achieve good results, producing relatively small file sizes and retaining the accuracy needed for the visualization without the accuracy loss being noticeable to the user, even though the improvement of the hybrid method compared to the others was not always as big in the best performing cases.

This work could serve as a base for several further improvements. The user interface could accept more parameters for more advanced usage and more advanced users wanting more fine-tuning in their simulations. More visualization options could be developed, such as volumetric rendering which requires the development of an efficient method to enable real-time rendering, and other 3D structures derived from streamlines, such as streamribbons.

The hybrid compression method could also see improvements in tweaking the B-Spline parameters and regression to enable a larger ratio of optimal B-Spline regressions among the octree blocks since its performance is limited in how many blocks are suitable for efficient B-Spline regression. Different volume subdivision methods could also be attempted to achieve better data distribution among individual subdivisions and limit the number of subdivisions unfit for B-Spline regression. Finally, the algorithm could be rewritten in a more efficient language and executed on a GPU to improve its execution speed.

Bibliography

[1] S. N. Doost, D. Ghista, B. Su, L. Zhong, Y. S. Morsi, Heart blood flow simulation: a perspective review, BioMedical Engineering OnLine 15 (1) (2016) 101. doi:10.1186/s12938-016-0224-8.

[2] C. A. Taylor, T. J. Hughes, C. K. Zarins, Finite element modeling of blood flow in arteries, Computer methods in applied mechanics and engineering 158 (1-2) (1998) 155–196.

[3] L. Antiga, M. Piccinelli, L. Botti, B. Ene-Iordache, A. Remuzzi, D. A. Steinman, An image-based modeling framework for patient-specific computational hemodynamics, Medical & biological engineering & computing 46 (11) (2008) 1097.

[4] Navier–stokes equations - wikipedia, the free encyclopedia, accessed: 2017-09-20.
URL https://en.wikipedia.org/wiki/Navier-Stokes_equations

[5] J. R. Womersley, Method for the calculation of velocity, rate of flow and viscous drag in arteries when the pressure gradient is known, The Journal of physiology 127 (3) (1955) 553–563.

[6] M. Zhou, O. Sahni, H. J. Kim, C. A. Figueroa, C. A. Taylor, M. S. Shephard, K. E. Jansen, Cardiovascular flow simulation at extreme scale, Computational Mechanics 46 (1) (2010) 71–82.

[7] Palabos - an open-source cfd solver based on the lattice boltzmann method, accessed: 2017-09-20.
URL http://www.palabos.org/software/lattice-boltzmann-method

[8] Sailfish - a free computational fluid dynamics solver based on the lattice boltzmann method, accessed: 2017-09-20.
URL http://sailfish.us.edu.pl/

[9] Openfoam - a free, open source cfd software, accessed: 2017-09-20.
URL http://www.openfoam.com/

[10] A. Updegrove, N. M. Wilson, J. Merkow, H. Lan, A. L. Marsden, S. C. Shadden, Simvascular: An open source pipeline for cardiovascular simulation, Annals of Biomedical Engineering 45 (3) (2017) 525–541. doi:10.1007/s10439-016-1762-8.

[11] Streamlines, streaklines, and pathlines - wikipedia, the free encyclopedia, accessed: 2017-09-20.
URL https://en.wikipedia.org/wiki/Streamlines,_streaklines,_and_pathlines

[12] S. K. Ueng, K. Sikorski, K.-L. Ma, Fast algorithms for visualizing fluid motion in steady flow on unstructured grids, in: Visualization, 1995. Visualization '95. Proceedings., IEEE Conference on, 1995, pp. 313–320. doi:10.1109/VISUAL.1995.485144.

[13] G. Anastasi, P. Bramanti, P. Di Bella, A. Favaloro, F. Trimarchi, L. Magaudda, M. Gaeta, E. Scribano, D. Bruschetta, D. Milardi, Volume rendering based on magnetic resonance imaging: advances in understanding the three-dimensional anatomy of the human knee, Journal of anatomy 211 (3) (2007) 399–406.

[14] Paraview - an open-source, multi-platform data analysis and visualization application, accessed: 2017-09-20.
URL https://www.paraview.org/

[15] Visualization toolkit - an open-source, freely available software system for 3d computer graphics, image processing and visualization, accessed: 2017-09-20.
URL https://www.vtk.org/

[16] Wavelet - wikipedia, the free encyclopedia, accessed: 2017-09-20.
URL https://en.wikipedia.org/wiki/Wavelet

[17] L. Belhadef, Z. M. Maaza, Lossless 4d medical images compression with motion compensation and lifting wavelet transform, International Journal of Signal Processing Systems 4 (2) (2016) 168–171. doi:10.12720/ijsps.4.2.168-171.

[18] G. Al-Khafaji, L. E. George, Fast lossless compression of medical images based on polynomial, International Journal of Computer Applications 70 (15).

[19] H. Lehmann, E. Werzner, M. A. A. Mendes, D. Trimis, B. Jung, S. Ray, In situ data compression algorithm for detailed numerical simulation of liquid metal filtration through regularly structured porous media, Advanced Engineering Materials 15 (12) (2013) 1260–1269. doi:10.1002/adem.201300129.

[20] Magnetic resonance in medicine: The basic textbook of the european magnetic resonance forum. 4th ed., Radiology 224 (1) (2002) 152–152. doi:10.1148/radiol.2233022512.

[21] C. R. Crawford, K. F. King, Computed tomography scanning with simultaneous patient translation, Medical Physics 17 (6) (1990) 967–982.

[22] J. M. Ollinger, J. A. Fessler, Positron-emission tomography, IEEE Signal Processing Magazine 14 (1) (1997) 43–55.

[23] Simvascular - manual model generation guide, accessed: 2017-09-20.
URL http://simvascular.github.io/docsModelGuide.html

[24] N. Wilson, K. Wang, R. W. Dutton, C. Taylor, A software framework for creating patient specific geometric models from medical imaging data for simulation based medical planning of vascular surgery, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2001, pp. 449–456.

[25] ParaViewWeb - a web framework to build applications with interactive scientific visualization inside the web browser.
URL https://www.paraview.org/web/

[26] IEEE standard for floating-point arithmetic, 2008, pp. 1–70. doi:10.1109/IEEESTD.2008.4610935.

[27] R. Sakai, D. Sasaki, S. Obayashi, K. Nakahashi, Wavelet-based data compression for flow simulation on block-structured cartesian mesh, International Journal for Numerical Methods in Fluids 73 (5) (2013) 462–476. doi:10.1002/fld.3808.

[28] D. Meagher, Geometric modeling using octree encoding, Computer graphics and image processing 19 (2) (1982) 129–147.

[29] Bezier curve - definition and basic properties, mit hyperbook, accessed: 2017-09-20.
URL http://web.mit.edu/hyperbook/Patrikalakis-Maekawa-Cho/node12.html

[30] G. Xu, C. Bajaj, Regularization of b-spline objects, Computer aided geometric design 28 (1) (2011) 38–49.

[31] Linear least squares - wikipedia, the free encyclopedia, accessed: 2017-09-20.
URL https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)

[32] Wavefront .obj file - wikipedia, the free encyclopedia, accessed: 2017-09-20.
URL https://en.wikipedia.org/wiki/Wavefront_.obj_file

[33] Node.js - a cross-platform javascript run-time environment for executing javascript code server-side, accessed: 2017-09-20.
URL https://nodejs.org/en/

[34] Numpy - the fundamental package for scientific computing with python, accessed: 2017-09-20.
URL http://www.numpy.org/

[35] H. Si, Tetgen, a delaunay-based quality tetrahedral mesh generator, ACM Trans. Math. Softw. 41 (2) (2015) 11:1–11:36. doi:10.1145/2629697.

[36] Parallel hierarchic adaptive stabilized transient analysis of compressible and incompressible navier stokes equations, accessed: 2017-09-20.
URL https://github.com/PHASTA/phasta

[37] MPICH - a high performance and widely portable implementation of the Message Passing Interface (MPI) standard, accessed: 2017-09-20.
URL https://www.mpich.org/

[38] Angularjs - a javascript-based open-source front-end web application framework, accessed: 2017-09-20.
URL https://angularjs.org

[39] Three.js - a cross-browser javascript webgl library/api, accessed: 2017-09-20.
URL https://threejs.org

[40] P. Lavrič, C. Bohak, M. Marolt, Collaborative view-aligned annotations in web-based 3d medical data visualization, in: 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2017, pp. 259–263. doi:10.23919/MIPRO.2017.7973430.

[41] P. Lavrič, A collaborative web based framework for visualisation of volumetric data, thesis, Accessed: 2017-09-18 (2016).
URL http://eprints.fri.uni-lj.si/id/eprint/3526